THE EDITOR

As mentioned earlier, source code in a FORTH system is stored in a group of blocks stored on disk or tape called 'screens'. When you go to edit a screen, the FORTH system first looks to see if it is already in memory. If not, it is called from the disk or tape unit first. While you are editing a screen, the changes being made are kept only on the display, the screen in memory is not being changed at the same time. When you exit the editor, you have the options of discarding the changes you made, updating the screen in memory, updating and saving it out to disk, or updating, saving it, and 'loading' (compiling) it.

To edit a screen, type:

<scr#> EDIT

If the requested screen # is not in memory, the screen will be read in from disk or tape. The first 1000 characters of the FORTH screen will be displayed. The remaining 24 characters of the FORTH screen will not generally be missed, but if necessary access to them is possible. All editing is done on the portion of the screen displayed. Editing a screen is done in much the same way as editing lines of a BASIC program. Several additional editing features have been provided to make the process even easier. Full cursor movement over the displayed screen is possible, however you are prevented from 'scrolling-up' the screen by going off the bottom. The color-select keys (<ctrl>-0, <ctrl>-1, etc.), cursor movement keys, RVS-ON, RVS-OFF, and HOME keys work normally. CLR-HOME does not clear the screen but performs the same function as HOME.

- SCREEN LINE LENGTH

Each line of the displayed screen is normally considered to be 40 characters in length. Inserting and deleting characters when in the middle of a screen line will cause the shifting of characters on that line only with the following two exceptions:

1) Function key 1 (f1) sets the current line to a length of 80 characters (the last displayed line cannot be set to a length of 80 characters). After this key is pressed, and up to when the current row changes via a cursor movement, pressing <return>, etc. the current line length will stay at 80. Any movement of characters on the current line from inserting or deleting characters will also cause shifting of characters on the following line, since it will be considered an extension of the current line.

2)If you are using the <del> key to delete the previous character, and you are currently at the first character of a line, the current line is then considered an extension of the previous line effectively making it 80 characters in length. The current line will be shifted into the previous line as you delete characters.

- BUFFERS USED BY EDITOR

Certain functions of the editor create and use buffers in the C64-FORTH memory space. The buffers are created within the space from above PAD up to the bottom of the disk buffers. An 80-byte CHARACTER buffer is used by the DELETE-CHARACTER-AND-SAVE function, and a LINE buffer is used by the DELETE-LINE-AND-SAVE function.

- INSERT functions

There are several different ways to insert text into the screen. First there is the insert key (activated by <shift>-DEL) on the 64 keyboard referred to as INS. INS causes all text starting from the current cursor position to the end of the line to be shifted to the right and a blank space is inserted at the current position. Text at the end of the line is 'shifted-off' and lost. Next, a character INSERT MODE is turned on by the <ctrl>-I key. Any further typing of alpha-numeric characters will expand the current line starting from the current cursor position and will insert the character typed. The insert-character function can be turned off by the <ctrl>-O key. There is also an INSERT-BLANK-LINE function. The function key f4 moves the rest of the screen down starting from the current line and inserts a blank line. As to what happens to the bottom line that got shifted off the screen, refer to the section 'THE 26th LINE'.

- LINE CLEAR functions

Two keys perform line clear functions. The <ctrl>-C key clears the current line and places the cursor at the start of the line. <ctrl>-V clears text on the current line starting at the current cursor position up through the last character of the line. Note that both of these functions work according to the currently set line length. Also, because of the 64 keyboard decoding, pressing the STOP key will function identically to <ctrl>-C in clearing the current line.

- DELETE functions

There are also several ways to delete text. The DEL key on the 64 keyboard deletes the character to the left of the current cursor position and shifts the rest of the line left one character position to take up the space. Spaces are added to the end of the line as it is shifted left. Note that a current line can be shifted into the previous line. The <ctrl>-D key deletes the current character and collapses the rest of the line to take up the space. Function key 3 (f3) deletes the current line. As with the INS key and the insert-character function, any text removed from the screen via the DEL key, the <ctrl>-D key, or f3 is lost forever. CUT and PASTE functions, described later, are used to delete and save characters and lines so that they may be moved to other locations in the current screen or to another screen.

- CUT and PASTE functions

Five functions are provided for 'cut and paste' operations. Four function keys, and one control key are used to provide: DELETE-CHARACTER-AND-SAVE (f7), DELETE-LINE-AND-SAVE (f5), UNDELETE-CHARACTER (f8), UNDELETE-LINE (f6), and 'POP'-A-LINE (<ctrl>-P). Characters and lines, as they are deleted, are saved off in temporary buffers. The f8 and f6 function keys pull out and insert into the current screen line the most recently saved characters and lines, if any were. The characters or lines are removed from the buffers as they are UNDELETED. The 'POP'-A-LINE key takes the most recently-deleted line from the line buffer and inserts it before the current line, but does not remove it from the line buffer. POP-A-LINE may be used to duplicate a line several times within one or more screens. The characters and lines saved in the temporary buffers may be passed to other screens AS LONG AS THE LENGTH OF THE FORTH DICTIONARY DOES NOT CHANGE! This is done by editing one screen, deleting characters and/or lines using the f7 and f5 keys, exiting the screen-editing mode, entering a second screen for editing, and using the f8 and/or f6 keys to insert characters and/or lines into the new screen. Between exiting the first screen and entering the editing mode of the second screen, no definitions may be added, deleted, or loaded, as this will cause loss of the saved characters and lines.

The character buffer in which characters deleted by f7 are saved in is 80 bytes (characters) long. When full, the delete-character function will stop working. The line-buffer uses any available space between the top of the FORTH dictionary and the bottom of the screen buffers area. Here too, when the space is filled, line-delete no longer works, and neither undelete-character nor undelete-line works when there are no characters or lines left in the buffers.

Note that the color of deleted characters and lines is not saved. Characters and lines are undeleted in the current color.

- THE 26th LINE

Where are the remaining 24 characters that aren't displayed? They reside in a 'mini-line' just below the bottom of the screen. Whenever any type of line-insert function is done, lines are pushed down the screen toward the bottom. The bottom line goes into the 26th line, the 'mini-line'. Whenever any type of line-delete function is done, lines are shifted up the screen and the 26th line is brought in and placed on line 25 of the screen. Since it is only 24 characters long, line 25 is padded with spaces on the screen. As line 26 is moved onto the screen, it is filled with spaces, although you can't see it. Also, when lines are shifted down the screen, the first 24 characters of line 25 go into line 26, the remaining 16 characters of line 25 are lost, as are the 24 characters that were in line 26.

Therefore, if you need all the space available in a screen, putting text in the last line is simple. Delete one of the lines currently on the screen using the DELETE-LINE function key (f5), which will save it in the line buffer. Move the cursor down to line 25 on the screen and type in the text that you want. Only the first 24 characters count. When set, move the cursor back up to where the line was deleted. Use the UNDELETE-LINE function key (f6) to fetch back the deleted line. The 24 character line at the bottom of the screen will disappear as lines are shifted down - it went into line 26. If you are worried about it being present, just use f5 again to delete lines and you'll see line 26 come up into the screen. Be careful inserting lines when you have text in line 26 as it will be shifted out and lost. Again, for most applications, screens are plentiful and to ignore the presence of the last 24 characters is no loss.

- NON-TEXT DATA FOUND IN A SOURCE SCREEN

The editor is for editing source text screens. Any non-text data found when the screen is first displayed is converted to a graphics character (usually the 'pi' character). When a screen is exited with update, the screen ASCII text is transferred into the screen buffer memory. Thus, if you create screens on disk which contain random byte data and not source text, and you accidentally enter the editing mode on one of those screens, you must exit WITHOUT UPDATE to preserve the data. The 64 has many control characters in it's display-control character set and allowing random byte data to be displayed would create a rather useless display. Hence, most non-text data bytes are displayed as a graphics character.

There are modes where control characters are accepted and displayed. Identical to BASIC, any control characters entered while in the 'quote' mode (i.e. within paranthesis) inserts a graphics representation of the control character. Also, any color-select key pressed while not in the quote mode enables the key selected, and the <shift>-COMMODORE key switches the case (the display font) as in normal BASIC editing mode.

- EXITING THE EDITOR AND SAVING OR DISCARDING WORK

There are four ways to exit the editor. <ctrl>-Z exits, and discards all work done. The contents of the screen in the disk buffer memory area is not in any way modified or marked as updated. <ctrl>-X exits the editor, updates the contents of the screen in memory, and marks that screen as updated. <ctrl>-F exits, the editor, updates the screen in memory, and goes through the disk buffer area and writes out to disk ALL screens that are marked as updated, which will include the screen just edited. The numbers of the screens are printed as they are saved. <ctrl>-L exits the editor, updates the screen in memory, saves all updated screens to disk, and begins loading the screen just the same as if you had typed: <scr#> LOAD. With <ctrl>-L or <ctrl>-F, if for any reason there is a disk or tape error while FORTH is trying to save the screens, the screens will be left as updated in memory, the error message will be printed, and control will return to the FORTH input routine. The problem causing the error can be corrected, then the screen(s) can be saved again by the FORTH command SAVE-BUFFERS.

EDITOR COMMAND SUMMARY

f1	- set current line to effective length of 80
f2	- set current line back to effective length of 40
f3	- delete current line (line is not saved)
f4	- insert a blank line
f5	- delete current line and save
f6	- undelete a line and insert at current row
f7	- delete current character and save
f8	- undelete a character and insert at current pos
INS	- insert a blank at the current cursor pos
DEL	- delete previous character
<ctrl>-B	- 'tab' (move cursor) 4 positions to the right
<ctrl>-C	- clear current line (RUN/STOP key does same)
<ctrl>-D	- delete current character (character not saved)
<ctrl>-I	- turn on auto-insert
<ctrl>-O	- turn off auto-insert
<ctrl>-P	- 'pick' the most recent line in the deleted-line buffer and insert before current line
<ctrl>-V	- clear to end-of-line
<ctrl>-Z	- exit editor, discard all work done this screen
<ctrl>-X	- exit editor, update screen in memory
<ctrl>-F	- exit editor, update screen, and save screens
<ctrl>-L	- exit editor, update screen, save all screens, and load this screen

THE ASSEMBLER

An assembler for generating 6502 machine code has been included in source code format with the C64-FORTH package. Once the assembler has been loaded into the FORTH dictionary, the entire FORTH program can be 'SAVESYSTEMed' and the assembler will be a permanent part of the bootable system. The assembler screens are screens # 6-17. To load the assembler, insert the C64-FORTH disk and type:

```
6 LOAD
```

Screen # 6, when through loading, automatically initiates loading of screen # 7, and so on until the entire assembler is created as part of the FORTH system. You may then save the new FORTH system onto a disk with the following command:

```
SAVESYSTEM "<filename>"
```

The assembler is provided in source code format because even though the 64 has approx. 51K of usable RAM when the BASIC ROMs are disabled, some C64-FORTH users may need every bit of available space. Once an application package has been written and debugged, the editor may no longer be needed. It consumes approx. 2.5K of memory, and it was placed as the last part of the C64-FORTH package. The editor may be deleted via 'FORGET EDITOR' freeing up the 2.5K. Since the assembler may be selectively loaded, the 2K of space that it consumes need not be taken up if the application package is written all in higher-level FORTH. Thus, you can customize your FORTH system with or without the editor, and with or without the assembler.

The assembler allows generating 6502 machine code directly from easily associated 'mnemonics', or instructions. Assembly language programs allow a one-to-one correspondence between program instructions and micro-processor operations. While FORTH produces very space-efficient and fast running code, some applications require the absolute control of the micro-processor that only machine code can give.

For users just picking up assembly language programming, the COMMODORE-64 PROGRAMMER'S REFERENCE GUIDE has a brief introduction on the subject, and there are several good books out on the market on programming the 6502 in assembly language. One useful point relative to C64-FORTH is that you don't need the 64MON cartridge for the 64 to write, enter, and execute machine language programs.

- CODING IN C64-FORTH ASSEMBLY

Coding in C64-FORTH assembly language is similar to coding in higher-level FORTH. Routines may originally be coded in FORTH and debugged, then recoded into assembly language with minimal changes or restructuring. Structure control directives (BEGIN---UNTIL, IF---ELSE---THEN, etc.), while implemented differently in the assembler vs. FORTH, function similarly. Therefore, to assure that assembler versions are used in assembly code, and to differentiate assembler mnemonics from some FORTH word definitions, all assembler control directives and mnemonics end with a comma (','). This also allows FORTH word definitions to be used during assembly to generate instruction addresses or to control the assembly process. For example:

```
CR ." ASSEMBLING BACKGROUND COLOR-SET ROUTINE"

  ( get current cursor color, and mask out high 4 bits
    using machine code AND instruction, not FORTH
    AND function )

646 LDA, HEX F # AND, DECIMAL              ( mask color value )
1 # CMP, 0= IF,                  ( is current cursor color = 1? )
  0 # LDA, 53280 STA,        ( yes, set border color to black  )
  53281 STA,                ( and set background color to black )
ELSE,                         ( current cursor color not white, )
  1 # LDA, 53280 STA,        ( then set border color to white  )
  53281 STA,                ( and set background color to white )
THEN,                         ( terminate conditional branches  )
RTS,                          ( return from this subroutine     )
```

In traditional assembly language code, a label is optionally specified first, then the instruction mnemonic, followed by the 'operand' (the address to operate upon). In FORTH, to ensure proper structured coding, labels are not permitted. Operands must preceed the instruction mnemonic, since in FORTH values and parameters used as qualifiers must be put onto the data stack before the routine that operates on them is called, and instruction mnemonics are actually defined as word definitions.

Note in the previous example the use of the HEX and DECIMAL word defs. The assembler does not COMPILE definitions, but is actually operating in the INTERPRETIVE mode, just like FORTH does after it prints 'OK' on the screen and waits for user input. The assembly code instruction mnemonics are fed into the input stream just the same as if you were typing them in in response to the 'OK' prompt. Therefore, the FORTH control structures, such as BEGIN---UNTIL , IF---ELSE---THEN , etc. cannot be used since FORTH is not compiling a definition. Instruction mnemonics are FORTH routines that execute immediately upon being read in. They generate and place into memory byte data that will effect the proper instruction.

You will also note in the table of C64-FORTH assembler mnemonics that there are none of the 6502 'branch' instructions. In order to encourage good programming technique, control directives must be used to effect branching, such as BEGIN,---UNTIL, , IF,---ELSE,---THEN, etc. (Note that the ASSEMBLER versions have a comma at the end of the directives. It is important to be aware of this distinction.) The control directive limitation has another advantage - while assembling source code, the assembler checks control directives for proper nesting. This helps ensure that program branching during run-time execution is at least predictable. An alternative to this would be to have the FORTH kernal control the execution of each instruction, and this would be prohibitive timewise.

- ACCESSING MACHINE CODE ROUTINES FROM HIGHER-LEVEL FORTH

The instruction mnemonics and structure control directives are all defined in the ASSEMBLER vocabulary. The normal way to generate machine code routines is by using the FORTH word definition 'CODE'. CODE automatically creates a FORTH header, establishes the linkage into the machine code required for access from higher-level FORTH routines, and sets the active (context) vocabulary to ASSEMBLER. Instruction mnemonics can then be used to add machine code to the definition. Returning to the higher-level FORTH routines must be properly done or the system will be corrupted (i.e. expect it to crash!). Finally, the definition must be verified for proper completion. This is done with the 'END-CODE' word definition.

Machine instructions within a machine code routine cannot call routines defined with the CODE/END-CODE structure. To call a machine code routine from another machine code routine the 'JSR' (mnemonic for the call-routine function) instruction is used, and to return back to the next instruction of the caller the 'RTS' (mnemonic for return-to-caller function) instruction must be executed. This call-return sequence is different from how a FORTH level definition calls or invokes a sub-definition.
There is another way that a FORTH level definition can access (call) a machine code routine. The CALL and CALLR FORTH

definitions allow directly executing machine code routines that terminate execution with the RTS instruction. These machine code routines may then be ROM routines within the 64 memory space or code routines generated via the C64-FORTH assembler. And, of course, any machine code routine that ends with a RTS instruction can be called from any other machine code routine.

Various tests are made within the assembler to detect user errors. CODE saves the data stack level at the start of assembly language definition. END-CODE checks to make sure the exiting level is the same as the starting level detecting missing or excess parameter type of errors. Structure control directives place check digits on the stack to ensure proper nesting and pairing. Unimplemented addressing modes and out-of-range values for operands are checked for and flagged. While these measures don't guarantee crash-free code, they do show up many of the more common errors. A little user caution is also necessary in becoming familiar with the assembler. For instance, the FORTH definition 0= does something entirely different than the ASSEMBLER definition 0= even though it functionally appears the same.

There are a number of 6502 instructions that are called 'implied' instructions. There are no operands required for these instructions, and assembly of the mnemonic generates a one-byte data item in memory that is the machine instruction. The implied instructions are:

BRK,	CLC,	CLD,	CLI,	CLV,	DEX,
DEY,	INX,	INY,	NOP,	PHA,	PHP,
PLA,	PLP,	RTI,	RTS,	SEC,	SED,
SEI,	TAX,	TAY,	TSX,	TXA,	TXS,
TYA,					

In the list of instructions available under the assembler, branch instructions (other than JMP,) are not included, as these are generated automatically by the structure control directives. The group of instructions left are arithmetic, logical, and data movement instructions that have more than one addressing mode each. Not every one of the following instructions have the same allowable addressing modes, so check a 6502 instruction reference if you're not sure. The multiple-mode instructions are:

ADC,	AND,	ASL,	BIT,	CMP,	CPX,
CPY,	DEC,	EOR,	INC,	JMP,	JSR,
LDA,	LDX,	LDY,	LSR,	ORA,	ROL,
ROR,	SBC,	STA,	STX,	STY,	

(Actually, the JSR, instruction above has only one addressing mode, but since it is a memory reference instruction, we'll let it pass.)

The 6502 has 11 addressing modes for memory-reference instructions. The 6502 considers the bottom page (256 bytes) of memory to be special, calling it the 'zero-page' (zp), and offers shorter, quicker instructions for accessing those locations. You do not have to specify zp mode in assembler source code - the assembler checks to see if the effective address is in the zero-page, and if so, generates the special zp instruction instead of the general memory reference instruction. This works for indexed via X or Y modes as well as absolute references. Before the switch to zp mode is made however, the assembler checks to see if the zp mode is allowed. If not, the general memory reference instruction is generated anyway.

- ADDRESSING MODE SPECIFICATION

The assembler generates a one, two, or three-byte data item in memory that is the machine instrution plus the effective address. Simple symbols inserted after the effective address and before the instruction mnemonic conveys to the assembler which mode is desired:

SYMBOL	DESIRED ADDRESSING MODE	OPERAND RANGE
.A	accumulator	none
#	immediate data	byte value
,X	indexed,X	zp or absolute
,Y	indexed,Y	zp or absolute
X)	indexed via X, indirect	zp only (byte adr)
)Y	indirect, indexed via Y	zp only (byte adr)
()	jump indirect	absolute adr only
none	direct memory reference	zp or absolute

Examples:

FORTH ASSEMBLER	CONVENTIONAL ASSEMBLER SOURCE CODE AND COMMENTS	
66 ADC,	ADC 66	;ADD ZP LOCATION 66
FFD2 JSR,	JSR FFD2	;CALL ROUTINE @ FFD2
.A ASL,	ASL A	;SHIFT ACC LEFT
5F # CMP,	CMP #5F	;IS A = 5F ?
TABLE ,X LDA,	LDA TABLE,X	;FETCH TABLE ENTRY + X
50 ,Y CMP,	CMP 50,Y	;CHECK BYTE IN ZP BUFFER+Y
0 X) LDA,	LDA (0,X)	;FETCH VAL INDIRECTLY + X
50)Y SBC,	SBC (50),Y	;SUBTRACT INDIRECTLY + Y
FFFC () JMP,	JMP (FFFC)	;JUMP TO RESET ENTRY POINT

- RETURNING TO HIGHER-LEVEL FORTH

Returning to the FORTH system must be done properly or the system will crash. The re-entry point into FORTH is actually the interpretive-pointer sequencer. This routine executes the code pointed to by the next pointer in the body of a FORTH definition. The assembler assigns the name 'NEXT' to the entry-point of this routine. To properly exit a machine code routine and re-enter FORTH, execute a JMP to NEXT.

Example:

```
( turn off auto-key repeat )
CODE RPTOFF
     0 # LDA,        ( clear loc. 650 to disable )
     650 STA,
     NEXT JMP,       ( return to FORTH )
END-CODE
```

Values may be added or removed to the data stack at the same time that control is returned to FORTH by jumping to other re-entry points. All of these routines perform the indicated function, then jump to NEXT for you. Refer to ASSEMBLER reference list for detailed descriptions.

PUT	-Overwrite bottom 16-bit value on data stack
PUSH	-Add 16-bit value to bottom of data stack
PUSH0A	-Add byte value to stack as 16-bit value
POP	-Remove bottom 16-bit value from data stack
POPTWO	-Remove bottom 2 16-bit values from data stack

- THE DATA STACK

The FORTH data stack is accessable from assembly code. The data stack is a section of zero-page memory 80 bytes in length. The top of the stack when empty starts at 6C. If one word value is on the stack, the top of the stack is at 6A, with the low byte of the value accessable at loc 6A, the high byte of the value is at loc. 6B. Values can be added to the stack until it is full, i.e. when the data stack pointer gets down to 1C. Whenever control returns to the FORTH input interpreter, and the stack pointer is down below 20 (hex), a stack-filled error is issued.

Assembly code may freely access, remove, and add data to the data stack. The minimum data item that may be added to the stack is a 16-bit value. The stack pointer must always be decremented or incremented by an even number when adding or removing data items. The stack pointer is actually the 6502 X register. The zero-page indexed-via-X addressing mode is used to access values on the stack. For example:

```
0 ,X LDA,       ( fetch low byte of bottom of stack )
```

Since the stack starts from a certain memory location and works down as values are added, the current position pointed to by the stack pointer is referred to as the BOTTOM of the stack. The TOP stack data-item referred to in FORTH is actually at the bottom of the stack. Words have been defined in the assembler vocabulary to make references to positions of certain data-items on the stack. Since the low byte of the most recently added data item on the stack is accessed via addressing mode 0 ,X , then the assembler definition 'BOT' sets that addressing mode for the current instruction.

For example:

```
BOT LDA,
```

will fetch into the accumulator the low byte of the first value on the stack. BOT 1+ LDA, will fetch into the accumulator the high byte of the first value on the stack. BOT 2+ LDA, will fetch the low byte of the second value on the data stack, BOT 3 + LDA, will fetch the high byte of the second value, and so forth.

Since the second data item on the stack is also frequently needed by a code routine, the symbol SEC has been provided. Thus,

```
SEC STA,
```

will store the contents of the 6502 accumulator into the low byte of the second data item on the stack. SEC is therefore equivalent to BOT 2+ or to the address specification 2 ,X , and SEC 1+ is equivalent to BOT 3 + , etc.

The C64-FORTH return stack is actually the 6502 machine stack. The 6502 uses page 1 of memory (from 100 to 1ff) for saving return addresses when subroutines are called. FORTH itself uses the return stack to save the interpretive pointer, to save the index and limit of DO loops, and to save temporary values. It is very easy to 'crash' the FORTH system and the 64 by improperly adding or removing values to the return stack. Only experienced FORTH programmers should attempt operations on the return stack (with the exception of some of the return operations to FORTH, as listed in RETURNING TO FORTH).

Values on the return stack may be obtained via the PLA instruction. The byte order convention is the same as with the data stack -i.e. low byte, then high byte. Therefore, to pull (remove) a 16-bit value from the return stack, PLA, TAY, PLA, will first remove the low byte of the value, save it in the Y register, then the high byte will be removed and saved in the accumulator.

To access values on the return stack without removing them, the S register of the 6502 is used. The S register is the machine stack pointer. It can be transferred into the X register, then any byte on the return stack may be accessed via the ,X addressing mode. Note that the X register must be saved and later restored before returing to FORTH. The process is:

1) save the X register in XSAVE

2) execute the TSX, instruction to bring the S reg to X

3) use 'RP)' as the address specification to address the lowest byte of the return stack. Use offsets to address higher bytes (RP) 1+ , RP) 2+ , RP) 3 + , etc.). The addressing mode is automatically set to ,X .

4) restore X before returning (load X from XSAVE)

The accumulator is not set to any particular value upon entry into a CODE routine. It may be freely used and does not have to be set to anything specific upon return to FORTH.

The X register is used as the data stack pointer by FORTH. If it is used for anything but accessing the data stack, it's value must first be saved, the operation performed, then it must be restored before returning to FORTH. Data items may be added to the stack by decrementing X twice for each item, or incrementing twice to remove each item.

Examples:

```
( push the value of 3 onto the data stack )
DEX, DEX,       ( make room for new entry  )
3 # LDA,        ( put 3 in low byte        )
BOT STA,
0 # LDA,        ( and clear high byte      )
BOT 1+ STA,

( set top of memory pointer from address on stack   )
BOT LDA, 643 STA,   ( fetch address and set pointer )
BOT 1+ LDA, 644 STA,
INX, INX,           ( remove item by incrementing X )

( set the screen cursor address from values passed on
  stack:    COL#  ROW#  ---
  NOTE: the X register is changed )
CODE SETCURSOR
  XSAVE STX,                              ( save X register )
  SEC LDY,                        ( set Y = desired col # )
  BOT LDA,                        ( set X = desired row # )
  TAX,
  CLC,                ( carry bit must be cleared to set )
  65520 JSR,                            ( call ROM routine )
  XSAVE LDX,                ( restore X before returning )
  POPTWO JMP,       ( remove values from stack & return )
END-CODE
```

The Y register is set equal to 0 upon entry into a CODE routine. It may be freely used and does not have to be restored before returning.

The S register, the 6502 machine stack pointer, points one byte below the low byte of the bottom return stack item. Improper alteration of the S register or values on the return stack will almost guarantee a system crash.

The 6502 is in binary mode upon entry to a CODE routine. If 6502 decimal calculations are done, binary mode must be reinvoked (via CLD,) before returning.

FORTH uses several variables while running to control execution of FORTH code. These variables are only accessable via machine code.

IP is the interpretive pointer. It points to the next pointer in the body of a FORTH word def. This will be the next word definition executed by FORTH when control is returned to it via jumping to NEXT.

W points to the code field of the FORTH definition just executed. W-1 contains the value 6C, which is the 6502 machine value for an indirect jump. FORTH enters a definition composed of machine code by pointing W to the code field address of the definition, which points to the machine code following it. FORTH jumps to W-1, the indirect jump instruction, the pointer is fetched out of the code field of the definition, and the 6502 is then executes the machine code of the definition.

UP is the User Pointer that contains the address of the base of the user area - a collection of variables defining the characteristics of the current FORTH system.

There also is a work space area used by FORTH that can be used by machine code routines. These are 9 consecutive bytes referenced by the assembler as N-1, N, N+1, ... N+7. These are zero-page memory locations, so they may be used as pointer locations for indirect addressing instructions, or as temporary storage locations. Traditionally N-1 is used for byte values, and N, N+2, N+4, and N+6 are used to hold 16-bit values. The 9 locations may be used freely. Values put into these locations may be passed to called subroutines, but not to other FORTH word definitions. Many of the routines in the FORTH kernal use the work space locations.

XSAVE is a byte location in zero-page memory that is used to save the X register during routines that need it for other purposes. Some of the examples in this section show how it is used.

- SETUP

SETUP is a subroutine in the FORTH kernal that is callable by a machine code routine. SETUP removes a specified number of 16-bit values from the data stack and stores them in the work area. To use the routine, the accumulator is loaded with the number of values to remove from the stack, then just call SETUP. The number of items to remove can only be from 1 to 4! The bottom stack item is put into N,N+1 , the second item is put into N+2,N+3 , etc.

Example:

```
3 # LDA, SETUP JSR,
```

In this example, the top 3 data-items are removed from the data stack and placed into N through N+5.

- CREATING MACHINE CODE SUBROUTINES

While within a CODE definition another CODE definition cannot be entered, a machine code subroutine that ends with the RTS, instruction can be called. CODE creates a definition header that is compatible with FORTH higher-level definitions, as well as setting up conditions that allow END-CODE to check for errors. We need something simpler just to create a header for a machine code routine. We can use 'CREATE' for this purpose. CREATE will create a header with the specified name. Assembly code may then be used to make up the body of the definition. The definition must end with either the RTS, instruction, or a JMP, to a machine code routine that terminates with the RTS, instruction. END-CODE is not needed to complete the definition, since no error-checking conditions were set up initially. When the name of the new definition is referred to, the ADDRESS of the first instruction in the body of the routine is placed on the data stack. Therefore, the NAME of the machine code routine may be used as an address specification in another machine code or CODE definition.

Example:

```
( create machine code routine first to return <> 0 if
  STOP key on 64 keyboard is depressed )
HEX
ASSEMBLER       ( select ASSEMBLER vocabulary since CODE
                  is not used )
CREATE STOPKEY?                                  ( create header )
  DC01 LDA,                          ( read in I/O port key bits )
  7F # CMP,                          ( if STOP pressed, then =7f )
  RTS,              ( return with Z status bit set or cleared )

( routine callable from FORTH waits until stop key pressed )
CODE WAITSTOP
  BEGIN,                                       ( start of wait loop )
    STOPKEY? JSR,                            ( see if STOP pressed )
  0= NOT UNTIL,                        ( loop until returns <> 0 )
  NEXT JMP,                                    ( return to FORTH )
END-CODE
```

A note of caution: Make sure you properly terminate a machine code routine. Since no error checking is done, the system could crash if a routine is called that was not terminated with a RTS, instruction or equivalent JMP, instruction.

- BRANCHING AND LOOP CONTROL STRUCTURES

As in higher-level FORTH, control structure directives must be used for program branching. The assembler versions of the directives effect the proper branch by placing into memory byte data corresponding to the 6502 branch instructions. They also place onto the data stack 'check' digits to help detect improper nesting of structures or incomplete structures.

The control structure directives are used for conditional looping or branching. The conditions allowed for branching are:

CONDITIONAL SPECIFIER	CONDITION TESTED
CS	is 6502 carry bit set ?
CS NOT	is 6502 carry bit clear ?
0=	is 6502 zero flag set ?
0= NOT	is 6502 zero flag clear ?
0<	is 6502 minus flag set ?
0< NOT	is 6502 minus flag clear ?
VS	is 6502 overflow flag set ?
VS NOT	is 6502 overflow flag clear ?

When a conditional specifier is used, a value is pushed onto the stack at assembly time that allows the directive that follows to place into memory the proper branch instruction to effect the testing of the condition.

CONDITIONAL FORWARD BRANCHING:

```
<conditional specification> IF,...ELSE,...THEN,
                            IF,...THEN,
                        (or IF,...ELSE,...ENDIF,
                        and IF,...ENDIF, )
```

As in higher-level FORTH, this structure is used for conditional forward branching. The condition is placed before the IF, part. Byte data is placed in memory such that at runtime if the condition tests TRUE, then control falls through to the first instruction after the IF, part. If not, a branch is taken to the first instruction after the ELSE, part (or THEN, if no ELSE, part was specified).

What the assembler version IF, actually does is remove the conditional placed on the stack, compile a branch instruction and temporary branch offset into memory, then place the current dictionary address onto the stack along with a check digit (the number '2'to indicate an IF,ELSE,THEN, type of structure). The way the stack is altered during assembly is:

```
conditional  ---  IFaddress  2
```

Remember that this action is performed during the ASSEMBLY process, not during run-time. During run-time, either the branch is taken, or the code following the branch is executed.

If an ELSE, directive is encountered in the input text, then first the check digit is checked to see that it is a '2' indicating the innermost conditional structure was started by an IF,. The address is then removed and the proper branch offset is calculated and placed into the branch instruction generated by the IF,. The current dictionary address is then placed onto the stack, along with a new check digit (also = 2). A JMP, instruction is placed into memory to allow the code executed by the IF, part to jump around the ELSE, part. A temporary jump destination address is placed in the instruction. The effect of the ELSE, during assembly is:

```
IFaddr  2  ---  ELSEaddr  2
```

Assembly code is assembled into memory until a THEN, directive is encountered. THEN, first checks the check digit to ensure it is a '2', then corrects the JMP, destination address if an ELSE, was part of the control structure, or corrects the branch offset generated by the IF, directive. The effect of THEN, during assembly is:

```
IFaddr  2  ---          or    ELSEaddr  2  ---
```

CONDITIONAL LOOPING:

```
BEGIN, ... <conditional specification> UNTIL,
                                       (or END, )
```

This structure is used for conditional looping or backward branching. The BEGIN, is used to mark the start of the loop, or the destination of the backward branch. Assembly code follows, then a condition is specified followed by the UNTIL, (or END,). The effect of this structure is to loop until a specified condition tests TRUE. Control then falls through, and the code following the UNTIL, part is executed.

BEGIN, only pushes the current dictionary address onto the stack followed by a check digit equal to 1. The effect of BEGIN, during assembly is:

```
---  BEGINaddr  1
```

UNTIL, first checks the check digit on the stack to verify that the inner-most control structure is a BEGIN,UNTIL, type of structure. Then, using the conditional and the address placed on the stack by BEGIN, , it generates a backward branch instruction in memory whose destination is the first instruction given after the BEGIN, directive. The effect of UNTIL, during assembly is:

```
BEGINaddr  1  conditional  ---
```

CONDITIONAL EXECUTION AND LOOPING:

BEGIN, ... <conditional specification> WHILE, ... REPEAT,

This structure is used primarily for executing code within a loop until a condition is met. The BEGIN, is used to mark the start of the loop. Assembly code may optionally follow, then a specified condition code is tested for. If the condition tests TRUE , then the code between the WHILE, and REPEAT, parts is executed. The REPEAT, causes an unconitional branch back to the first instruction after the BEGIN,. If the condition tests FALSE, then the loop is exited and the first instruction specified after the REPEAT, is executed.

BEGIN, only pushes the current dictionary address onto the stack followed by a check digit equal to 1. The effect of the BEGIN, during assembly is:

--- BEGINaddr 1

When the WHILE, directive is encountered, first the most recent check digit on the stack is checked to see that it is a '1' indicating the innermost control structure was started by a BEGIN,. The equivalent to an IF, directive is executed to place a conditional forward branch instruction into memory, however a 4 is left on the stack as a check digit. The effect of the WHILE, during assembly is:

BEGINaddr 1 conditional --- BEGINaddr 1 WHILEaddr 4

When the REPEAT, directive is encountered, the second check digit on the stack is checked to insure it is a '1' indicating this whole mess was started with a BEGIN,. A JMP, instruction is placed into the dictionary at this point to jump back to the first instruction after the BEGIN, part. Next, the first check digit on the stack is checked to insure it is a '4' indicating a WHILE, was specified. The equivalent to a THEN, directive is executed to terminate the conditional forward branch instruction placed into memory by the WHILE, part. The effect of the REPEAT, during assembly is:

BEGINaddr 1 WHILEaddr 4 ---

Examples:

```
( Wait for a key to be pressed - if key pressed was 'L' then
  put a TRUE flag on the stack, else, put a FALSE flag on
  the stack. )
HEX
CODE ?LKEY
  XSAVE STX,                                ( save X register )
  BEGIN,                                ( start of key-input loop )
    FFE4 JSR,                 ( get a value from keyboard buffer )
  0= NOT UNTIL,                             ( wait until <> 0 )
  4C # CMP,                             ( is 'L' key pressed ? )
  0= IF,
    1 # LDA,                        ( yes, put true flag in acc )
  ELSE,
    0 # LDA,                        ( no, put false flag in acc )
  THEN,
  XSAVE LDX,                              ( restore X register )
  PUSHOA JMP,     ( push flag onto stack & return to FORTH )
END-CODE


( move sprite #0 down the screen until it either reaches
  raster line # 150 or collides with another sprite )
HEX
CODE DOWNSCRN
N 1- STY,    ( use N-1 as terminating flag - zero it first )
BEGIN,                                            ( start of loop )
  D001 LDA,     ( is sprite #0 at or beyond line 150 yet ? )
  DECIMAL 150 # CMP, HEX
  CS IF,
    1 # LDA, N 1- STA,          ( yes, set flag to terminate )
  THEN,
  D01E LDA,    ( has sprite collided with another sprite ? )
  1 # AND,
  N 1- ORA,                             ( 'or' in terminate flag )
0= WHILE,                                  ( exit if result <> 0 )
  D001 INC,    ( else, increment row position of sprite #0 )
REPEAT,                                ( loop back and continue )
NEXT JMP,                                    ( return to FORTH )
END-CODE
```

As referred to in the section BRANCHING AND LOOP CONTROL STRUCTURES, the assembler checks for proper nesting of control structures and terminates with an error condition if improper nesting is discovered. What if, for instance, you want to forward branch out of a BEGIN,...UNTIL, loop upon a certain condition? There is a way to 'trick' the assembler into generating the proper branch instructions by manipulating the check digits placed on the stack by the control directives.
In the above case, the control structure you want to implement is:

BEGIN, ... <conditional> IF, ... <conditional> UNTIL, THEN,

Following the previously described course of assembly for the directives, the condition of the data stack as the IF, directive is encountered is:

BEGINaddr 1 conditional --- BEGINaddr 1 IFaddr 2

Since the innermost structure is a IF,THEN, type of structure, and the next directive encountered is UNTIL, , you must trick the assembler into thinking the innermost structure is a BEGIN,UNTIL, type. Both the check digits AND the addresses placed onto the stack must be switched. 2SWAP works very nicely for this purpose. 2SWAP must be used AFTER the IF, , and before the conditional specification of the UNTIL, directive. After the 2SWAP is executed, the data stack will look like:

BEGINaddr 1 IFaddr 2 --- IFaddr 2 BEGINaddr 1

Then when the <conditional> UNTIL, is encountered, the proper backward branch is placed into memory, and when the THEN, is encountered, the conditional forward branch created by the IF, is properly terminated.

Example:

```
( move sprite #0 down one raster line at a time until it
  either hits another sprite or a background pattern )
HEX
CODE SPRITEDOWN
BEGIN,                                          ( start of loop )
  D001 INC,                ( move sprite #0 down 1 raster line )
  D01E LDA,     ( has sprite #0 collided w/another sprite ? )
  1 # AND,
  0= IF,                           ( if yes, branch out of loop )
    D01F LDA,  ( no, has sprite collided with background ? )
    1 # AND,
2SWAP                        ( switch addresses & check digits )
0= NOT UNTIL,                                ( branch back if not )
THEN,                                ( terminate forward branch )
NEXT JMP,
END-CODE
```

So you've heard all your assembly-language programmer friends extol the virtues of using MACROS? And, you figure, if they can use them in their 'legitimate' programming, why can't you use them in FORTH ASSEMBLER? Go ahead, we're not stopping you. MACRO capability in FORTH happens not to be a special added feature, but an allowable function just by default.

MACROS in assembly programming are a way of defining a sequence of source code instructions to be generated just by specifying the name of the MACRO and perhaps a qualifier or two. Sounds similar to the way normal higher-level FORTH works? You got it.

Since the FORTH assembler is comprised of definitions that execute upon 'source' text being read in, you can create a new definition that causes several previously-defined definitions (e.g. assembler mnemonics) to execute just by specifying the new definition. Also, since the assembler words use the data stack to calculate the effective addresses of machine code instructions, parameters or qualifiers may be passed to the new definition to customize the generated code.

You'll be wanting use of macros only when using the assembler, so it's best to define macros as part of the assembler vocabulary. This way macro definitions won't be searched of listed when doing normal higher-level FORTH work. When you are done defining macros, select the vocabulary you wish to add new definitions to. The procedure is:

```
ASSEMBLER DEFINITIONS

< macros are defined >

FORTH DEFINITIONS

< new CODE definitions can be created via the assembler
  but placed in the FORTH vocabulary >
```

A common example of a commonly desired macro is to increment a 2-byte (16-bit) pointer. Since the 6502 is a 8-bit machine, to do this in normal FORTH assembly code, using N+2 as the pointer location for this example, is:

```
N 2+ INC, 0= IF, N 3 + INC, THEN,
```

This code sequence must be used every time a pointer is to be incremented in the assembly code program. A macro can be defined as such:

```
ASSEMBLER DEFINITIONS
: INC2,  DUP INC, 0= IF, ROT 1+ INC, THEN, ;
FORTH DEFINITIONS
```

Then within the source code, the macro is invoked by:

```
. . .  N 2+ INC2,  . . .
```

Analysis of the macro INC2, will show that the proper code will be generated. Other examples of macros are:

```
ASSEMBLER DEFINITIONS

( clear the accumulator      ---    )
: CLRA, 0 # LDA, ;

( push the location of a pointer onto stack     addr  ---  )
: PSHLOC, DUP 1+ LDA, DEX, BOT STA, LDA, DEX, BOT STA, ;

( decrement a 16-bit counter     addr  ---    )
: DEC2, DUP LDA, 0= IF, 3 PICK 1+ DEC, THEN, DEC, ;

( push a pointer onto the stack & return to FORTH
  addr  ---   )
: PSHNEXT, DUP LDA, PHA, 1+ LDA, PUSH JMP, ;

( the following does the same thing as the one just defined,
  but shows example of nesting of macros )
: PSHNEXT, PSHLOC, NEXT JMP, ;
```

#

Sets 'immediate' addressing mode for the next instruction.

)Y

Sets 'indirect indexed via Y' addressing mode for the next instruction.

,X

Sets 'indexed via X' addressing mode for the next instruction.

,Y

Sets 'indexed via Y' addressing mode for the next instruction.

.A

Sets the accumulator direct addressing mode for the next instruction.

0<

Conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'N' status bit is set as the result of some previous operation.

0=

Conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'Z' status bit is set as the result of some previous operation.

;CODE

Used as a way to conclude a colon definition with machine code to create a new type of FORTH level definition. The new type is defined as:

: <typename> ...FORTH defs... ;CODE ...assembly code...

;CODE stops compilation and terminates the definition <typename> by compiling (;CODE) into definition. The CONTEXT vocabulary is set to ASSEMBLER, and the following assembly code is added to the definition. Note: at least one FORTH definition must be called between <typename> and ;CODE.

When <typename> is later used to define a new word as in:

<typename> <defname>

the word <defname> will be created with its execution procedure given by the machine code following ;CODE. That is, when <defname> is executed, the machine code specified after the ;CODE of <typename> is executed.
(;CODE is actually defined in the FORTH kernal. The assembler redefines it as:
: ;CODE [COMPILE] ASSEMBLER [COMPILE] ;CODE ; IMMEDIATE
which will make the ASSEMBLER the CONTEXT vocabulary before invoking ;CODE)

ASSEMBLER

Make the ASSEMBLER vocabulary the CONTEXT vocabulary. It will be searched first as each word is encountered in the input stream for a match.

BEGIN,

A structure control directive thats marks the destination address for a backwards loop. Used with either the UNTIL, directive or the WHILE, and REPEAT, directives, as in:

BEGIN, ... <conditional> UNTIL, or
BEGIN, ... <conditional> WHILE, . . . REPEAT,

BOT

An assembler symbol that sets the addressing mode to access the bottom byte of the data stack. An positive offset may be added to this symbol to access other bytes on the data stack. Used as follows:

BOT LDA, BOT 1+ STA, BOT 5 + ADC, etc.

CODE

A defining word used in the form:

CODE <defname> . . . . . END-CODE

to create a dictionary entry for <defname> in the CURRENT vocabulary. <defname>'s code field contains the address of its parameter field. When <defname> is later executed, the machine code in this parameter field will execute. The CONTEXT vocabulary is made ASSEMBLER to make available the assembly code instruction mnemonics and directives.

CS

A conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'C' status bit is set as the result of some previous operation.

ELSE,

A structure control directive that marks the end of the code to be executed if a status condition tested TRUE , and marks the start of the code to be executed if that status condition tested FALSE. Used in the following way:

```
<conditional>  IF, . . . ELSE, . . . THEN,
```

END-CODE

Used to mark the end of a CODE definition. END-CODE checks Checks that stack position is same as when CODE was executed to check for imbalanced directives or parameters. Then the most recently defined CURRENT vocabulary definition is unsmudged making it available for execution from FORTH. END-CODE exits the ASSEMBLER making CONTEXT the same as CURRENT. This word was named ;C in early versions of FORTH.

IF,

A structure control directive that creates a forward branch instruction based on a 6502 status condition. At runtime, if the status tests TRUE, execution falls through to the following code. If it tests FALSE, the processor branches. Used as follows:

```
<conditional>  IF, . . . ELSE, . . . THEN,    or
<conditional>  IF, . . . THEN,
```

IP

Assembler symbol that specifies the address of the FORTH interpretive pointer. Used as follows:

```
IP LDA,   IP )Y LDA,    etc.
```

At run-time, the NEXT routine moves IP ahead within a colon-definition. Therefore, IP points just after the execution address being interpreted. If an in-line data structure has been compiled (e.g. a character string), indexing ahead by IP can access this data.

N

Assembler symbol that specifies the address of a 9-byte workspace in zero-page memory. Within CODE routines and machine code subroutines, free use of the workspace locations may be made over the range N-1 to N+7.

NEXT

Assembler symbol that specifies the address of the normal

re-entry point in FORTH from a CODE definition. All CODE definitions must return to NEXT or to a routine that returns to FORTH (PUSH , PUT , PUSHOA , POP , POPTWO).

NOT

During assembly, NOT reverses the previous conditional specification. When the following control directive is encountered, a conditional branch instruction is generated that tests if the specified 6502 status bit is clear. For example:

```
CS NOT IF,
```

will, at run-time, fall through to the following code if the carry status bit is cleared, or will branch if the carry bit is set.

POP

Assembler symbol that specifies the address of the re-entry point in FORTH that will remove and discard the bottom data stack entry and then jump to NEXT.

POPTWO

Assembler symbol that specifies the address of the re-entry point in FORTH that will remove and discard 2 data stack entries, then jump to NEXT.

PUSH

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, push a new data-item onto the data stack. The contents of the accumulator will be used as the high byte of the new data-item, and the bottom byte of the return stack will be fetched and used as the low byte of the new item. For example, to return to FORTH and at the same time push the contents of N, N+1, :

```
. . . N LDA, PHA, N 1+ LDA, PUSH JMP, END-CODE
```

PUSHOA

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, push the contents of the accumulator onto the data stack, then jump to NEXT. The value in the accumulator is placed as the low byte of the new stack entry, the high byte of the new entry is set equal to 0. Example:

```
. . . 1 # LDA, PUSHOA JMP, END-CODE
```

will return and push a value of 1 onto the data stack.

PUT

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, replace the current bottom entry on the data stack with the contents of the accumulator and the bottom of the return stack. The accumulator is used as the high byte, the bottom byte on the return stack is fetched and used as the low byte. Example:

```
. . . N LDA, PHA, N 1+ LDA, PUT JMP, END-CODE
```

REPEAT,
A structure control directive that creates an unconditional jump instruction to the address left on the stack by a BEGIN, directive. Used in the following way:

```
BEGIN, . . . <conditional> WHILE, . . . REPEAT,
```

RP)
Assembler symbol that specifies the addressing mode required to access the bottom byte of the return stack. For this to work at run-time, the return stack pointer (the S register) must have been transferred to the X register). See section called 'RETURN STACK'.

SEC
Identical to BOT 2+ . Assembler symbol that specifies the address and addressing mode necessary to access the low byte of the second entry on the data stack. Used as follows:

```
SEC LDA,     SEC 2+ STA,   etc.
```

THEN,
A structure control directive that terminates a forward branch instruction created by either an IF, or an ELSE, control directive. Used as follows:

```
<conditional>  IF, . . . ELSE, . . . THEN,    or
<conditional>  IF, . . . THEN,
```

UNTIL,
A structure control directive that creates a conditional branch instruction that will branch to the address defined with a BEGIN, directive if the tested condition is FALSE. Program execution falls through to the following code if the condition tests TRUE. Used with BEGIN, to create a program loop or backward branch. Used as follows:

```
BEGIN, . . .  <conditional specification> UNTIL,
```

UP
Assembler symbol that specifies the address of the pointer to the base of the USER AREA. The user area is a block of variables that define the characteristics of the current FORTH system. Used as follows:

```
UP LDA,   UP )Y LDA,  etc.
```

W

Assembler symbol that specifies the address of the pointer to the code field (execution address) of the FORTH definition begin executed. Indexing from the W pointer can access any byte in the definition's parameter field. For example:

```
2 # LDY, W )Y LDA,
```

will fetch the first byte of the parameter field of the CODE definition executing these instructions.

WHILE,

A structure control directive that creates a forward branch instruction based on a 6502 status condition. Used between the BEGIN, and REPEAT, directives, WHILE, allows a way to exit a program loop. The branch is taken if the tested condition is FALSE. If it tests TRUE, execution falls through to the next instruction specified after the WHILE,. Used as follows:

```
BEGIN, . . .  <conditional> WHILE, . . . REPEAT,
```

X)

Sets 'indexed via X, indirect' addressing mode for the next instruction.

XSAVE

Assembler symbol that specifies the address of a zero page location that can be used for temporarily saving the contents of the X register. Since the X register indexes to the data stack in zero page, it must be saved and restored when used for other purposes. Used as follows:

```
XSAVE STX,  . . .  XSAVE LDX,
```

FORTH GRAPHICS ON THE COMMODORE 64

The graphics extensions avaiable with C64-FORTH are not part of the normal FORTH system, but are an already compiled object code block loaded in and linked into the system. LIST screen 1 to see which screen to start loading to do this. There will be two options for loading the graphics - one will load them after the floating-point routines in the base FORTH system, the other alternative is to load them OVER the floating-point routines. If your program will not need real number capability, then you can save space by placing the graphics where the f.p. routines were. Note that to add the graphics extensions to the system, you must start with an unexpanded copy of the FORTH just as it boots up from the original C64-FORTH disk. You cannot add the graphics extensions if you have made ANY changes to the original FORTH system. Once the graphics extensions are added, you may then extend or customize the FORTH system to your needs, do a SAVESYSTEM to save it as a new system, etc.

To use the graphics extensions, you MUST be familiar with how the different graphics modes of the Commodore 64 work. A thorough reading of the section on graphics in the Commodore 64 Programmer's Reference Guide is a must. The following Graphics Glossary contains description of all the usefull words in the graphics extensions. You will also find several graphics demo programs on the C64-FORTH disk. Analyzing these will help you understand the function of the differenct words.

A bit-map display may be set up in either hi-res or multi-color mode. All the pixel-modifying routines, such as LINE, ARC, BOX, etc. work according to which mode is currently set. All the user has to do is select or deselect multi-color mode via the SETMC or CLRMC commands.

VIDMEM must be used to initialize the VIC-chip to display in certain memory ranges. If the character mode is selected, then the addresses of the character ROM and the screen memory must be specified. Since the VIC-chip operates in one of four 'banks' (16K ranges), the highest two bits of the address passed for the character ROM are used to set the bank #. Thats why in examples you'll see the routine to reset the VIC-chip to normal operation as:

```
HEX  1000  400  VIDMEM
```

Its easy to understand why 400 is passed as the address of the screen memory, but you may wonder why 1000 is passed as the address of the character ROM, when it is at absolute address D000 underneath the address space of the VIC-chip. The reason is that the VIC-chip is normally set to bank 0, so the two highest bits of the character ROM address must be 0's.

VIDMEM also sets two variables - MBASE and SBASE. MBASE is used by some routines to obtain the base address of the bit-map memory, so a specific pixel location can be calculated. SBASE is used to hold the address of the base of the screen memory in character mode, or the color memory in bit-map mode.

The bit-map graphics words draw within a specified window. The drawing window is defined by the variables, XLOW, YLOW, XLIM, and YLIM. The largest size of the window may be 0,0 to 319,199 (159,199 in multi-color mode). If the variables are set so that the window extends outside this range, this range is used as the window.

SETBM and SETMC both initialize these variables, so if you wish to redefine the window, you must do it after using these words to define the initial graphics modes. The order in which words whould be executed then to set up a graphics screen is:

VIDMEM, SETBM/CLRBM, SETMC/CLRMC, then alter the window variables XLOW, YLOW, XLIM, and YLIM.

Once the display is set up, you may switch between different display screens by using just VIDMEM. The other words are not needed once the VIC-chip is properly set. You only need VIDMEM to change the addresses that point to display areas.

The RAM underneath the C64 kernal ROMs may be used as freely for bit-mapped graphics as any other RAM. The pixel-modifying routines automatically switch in & out the ROMs as needed. The STOP/RESTORE key function will work properly even if the kernal ROMs happen to be switched out at the time of the keys being pressed, but RS-232 input or output happening coincidently may be lost or garbled.

GRAPHICS GLOSSARY

The following glossary contains descriptions of the graphics functions provided. It follows the same format as the FORTH glossary at the end of this manual. None of the words are part of any FORTH standard, and only the words that are of possible use to the programmer are described.

Most of the FORTH level graphics words merely call subroutines written in assembly language. The machine code equivalents of the higher-level FORTH words have the same name preceeded by an asterisk. In this glossary, if a FORTH level word merely calls a machine code subroutine, the name of the subroutine will given in parenthesis at the end of the description of the word. Assembly language programmers may program graphics with the same ease as in higher-level FORTH.

ARC Strtangle Endangle Radius ---

Draw an ARC or CIRCLE from starting angle specified to ending angle specified, with the radius specified. The center of the ARC or CIRCLE drawn is about the pixel value found in CURX,CURY. ARC drawing is always done counter-clockwise. A circle is drawn by specifying the starting angle equal to the ending angle, or the ending angle 360 degrees greater than the starting angle. The range for the angles is 0 <= angle <= 360 degrees. (*ARC)

Example of a 3/4 arc of radius 50 is:

```
0  270  50  ARC
```

Example of a full circle of radius 90 is:

```
0  0  90  ARC    or    0  360  90  ARC
```

BCLR X Y ---

Turn on clear-bit mode and write the pixel. (*BCLR)

BOX DeltX Delty ---

Draw a box starting from the current location in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+Delty. (*BOX)

BSET X Y ---

Turn on set-bit mode and write the pixel. (*BSET)

BXOR X Y ---

Turn on exclusive-or mode and write the pixel. (*BXOR)

CBASE --- adr

A variable that contains the base address of the character ROM. Normally set to D000, the user may change this if the character set has been redefined in RAM. Used only by RCHR, the routine that writes a ROM character into the hi-res screen.

CLRBM ---

Turns off bit-map mode.

CLRBMM ---

Clears the bit-map memory.

CLRMC ---

Turns off multi-color mode.

COSINE ang --- cosine

Returns a signed 16-bit integer that is the cosine of the angle specified. The angle specified should be in the range 0 <= angle <= 360 degrees. (*COSINE)

CURX --- adr

A variable that the user sets with the X value of the pixel to start drawing at (in the case of LINETO, RLINETO, BOX, or RECT) or the X value of the center of the arc or circle to draw. CURX will contain the X value of the last pixel drawn upon return from LINE, LINETO, RLINETO, BOX, or RECT, or if ARC is called, will contain the X value of the center as specified when the ARC routine was called.

CURY --- adr

A variable that the user sets with the Y value of the pixel to start drawing at (in the case of LINETO, RLINETO, BOX, or RECT) or the Y value of the center of the arc or circle to draw. CURY will contain the Y value of the last pixel drawn upon return from LINE, LINETO, RLINETO, BOX, or RECT, or if ARC is called, will contain the Y value of the center as specified when the ARC routine was called.

FILBMM colorval ---

Fills the bit-map memory with the 8-bit value passed.

FILCLR colorval ---

Fills the color memory with the 4-bit value passed.

FILSCRN colorval ---
Fills the screen memory with the 8-bit (2-nybble) value passed.

GETADR --- X Y
The current values in CURX and CURY are fetched and pushed onto the stack. (*GETADR)

GETCEN --- X Y
GETCEN returns on the stack the pixel address of the center of the current drawing window. (*GETCEN)

GFLGS --- adr
Variable that is used as flag bits to indicate various graphics modes. Can be ignored when using FORTH level graphics utilities, but the machine language programmer must set/clear certain bits for the pixel-writing routines to work properly. Bit 15 is set if multi-color pixel writing is desired, cleared for normal hi-res pixel writing. Bit 7 is set to have the pixel-writing routine write in set-bit mode, Bit 6 is set to have it write in clear-bit mode, and both bits 6 & 7 must be cleared to have it write in the exclusive-or mode. All the graphics routines that write to the multi-color or hi-res screen call one pixel-writing routine, and these bits must be properly set for the desired writing mode.

GRCLR ---
Turn on clear-bit mode. All subsequent pixel drawing will be done in clear-bit mode.

GRSET ---
Turn on set-bit mode. All subsequent pixel drawing will be done in set-bit mode.

GRXOR ---
Turn on exclusive-or mode. All subsequent pixel drawing will be done in exclusive-or mode.

LASTX --- adr
A variable that is updated when a pixel in memory is actually modified. When a pixel-writing routine (such as LINE) returns, LASTX will contain the X value of the last pixel that was written. For instance, if a line is being drawn that extends out of the current drawing window, CURX will return with the X value of the ending pixel of the line, but LASTX will contain the X value of the last pixel that was drawn within the current drawing window.

LASTY --- adr

A variable that is updated when a pixel in memory is actually modified. When a pixel-writing routine (such as LINE) returns, LASTY will contain the Y value of the last pixel that was written. For instance, if a line is being drawn that extends out of the current drawing window, CURY will return with the Y value of the ending pixel of the line, but LASTY will contain the Y value of the last pixel that was drawn within the current drawing window.

LINE strtX strtY endX endY ---

Draw a line from strtX,strtY to endX,endY. CURX and CURY left at endX,endY. (*LINE)

LINETO endX endY ---

Draw a line from the current pixel location in CURX,CURY to endX,endY. CURX,CURY updated to endX,endY. (*LINETO)

MBASE --- adr

Variable set by VIDMEM to point to the starting address of the bit-map memory.

MCCLRVAL N --- where 0 <= N <= 3

Sets the 2-bit value to write to multi-color bit-map memory when in clear-bit mode.

NOTE: If exclusive-or drawing function is selected (via GRXOR) in multi-color mode, then the pixel-modifying algorithm checks first to see if the 2-bit pair at the current pixel address is equivalent to the 2-bit value passed via MCSETVAL. If so, the 2-bit value passed via MCCLRVAL is placed in the current pixel location; otherwise, the 2-bit value passed via MCSETVAL is placed in the current location.

MCSETVAL N --- where 0 <= N <= 3

Sets the 2-bit value to write to multicolor memory when in clear-bit mode (see note under MCCLRVAL).

PIXMSK X Y --- adr msk

For a given X,Y location, PIXMSK returns on the stack the address of the byte in the bit-map memory that contains the pixel, and the mask that can be used on the byte, if read, to isolate the pixel value itself. (*PIXMSK)

RCHR chr# X Y ---

Draws a ROM character into the bit-mapped memory with the upper-left corner of the character starting at X,Y. The character is zoomed (enlarged) according to the current value in the variable ZOOM. Chr# is the same value as if you were POKEing a character into the screen memory - i.e. if you wanted a capital "A", chr# would equal 1. There are tables of the screen display codes in both the Commodore 64 User's Guide and the Commodore 64 Programmer's Reference Guide. (*RCHR)

RDPIX X Y --- N

Returns the current value of a pixel. If the screen is in hi-res mode, the value N returned will be in the range 0-1, if the screen is in multi-color mode, the value returned is in the range 0-3.

RECT Deltx Delty ---

Draw a filled-box starting from the current location in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+Delty. (*RECT)

RLINETO Deltx Delty ---

Draws a line in relative mode from the current pixel address in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+DeltY. (*RLINETO)

RSPXY Sprite# DeltX DeltY ---

Moves the specified sprite a RELATIVE amount. DeltX is added to the current X position of the sprite, and Delty is added to the current Y position of the sprite. (*RSPXY)

SBASE --- adr

Variable set by VIDMEM to point to the starting address of the screen memory.

SETADR X Y ---

Sets the current address. Takes the top two stack values and places them in CURX,CURY. (*SETADR)

SETBM ---

Turns on bit-map mode.

SETBMM ---

Sets all bits in the bit-map memory high.
(eq. to 255 FILBMM)

SETMC ---

Turns on multi-color mode.

SINE ang --- sine

Returns a signed 16-bit integer that is the sine of the angle specified. The angle specified should be in the range 0 <= angle <= 360 degrees. (*SINE)

SP1X SP# ---

Sets the specified sprite # to 1X magnification (non-expanded in the X direction).

SP1Y SP# ---

Sets the specified sprite # to 1Y magnitication (non-expanded in the Y direction).

SP2X SP# ---

Sets the specified sprite # to 2X magnification (expanded in the X direction).

SP2Y SP# ---

Sets the specified sprite # to 2Y magnification (expanded in the Y direction).

SP<BKGD SP# ---

Sets the priority of the specified sprite to be less than the background.

SP>BKGD SP# ---

Sets the priority of the specified sprite to be greater than the background.

SPCOLOR SP# COLOR ---

Sets the sprite specified to the color specified.

SPDEFINE --- (at compile time)
--- adr (when created def is executed)

SPDEFINE is used to create a new dictionary entry, read in 63 consecutive ASCII representations of numbers, converts them to binary form, and stores them in the new definition. This word is provided for defining sprites. When the name of the new definition is later executed, the address of the first byte of the sprite data is pushed onto the stack. The word SPMOV may be used to move the data up to an accessible area in display memory. When either HEX or BINARY (2 BASE !) mode is used, this is a simple way to convert a graph-paper representation of a sprite figure into binary code. The graphic demos on the C64-FORTH disk contain some examples of how to use this word to define sprites.

SPMCOFF SP# ---

Turns off multi-color mode for the specified sprite.

SPMCON SP# ---

Turns on multi-color mode for the specified sprite.

SPMOV fromadr toadr ---

Moves a 63 byte chunk of data from the address 'fromadr' to address 'toadr'. The 'toadr' is rounded down by SPMOV to the nearest 64-byte boundary before the data is moved. Intended for moving a sprite definition from the FORTH dictionary up to the a memory range in the current video display bank.

SPMSK SP# --- mask

When passed a sprite # in the range 0-7 a mask is returned that corresponds to the binary power of the sprite #. This means that a sprite # of 0 returns 1, a sprite # of 1 returns 2, etc. The mask may be used to mask out unused bits in a VIC-chip register to look for something related to just one sprite. For example, to check if sprite #3 has collided with the screen background data, execute:

```
3 SPMSK D01F C@ AND
```

and a non-zero flag will be left on the stack if a collision occured. To check if any of sprites 0, 4, and 7 have collided with the background, execute:

```
0 SPMSK 4 SPMSK OR 7 SPMSK OR D01F C@ AND
```

SPOFF SP# ---

Turns off the specified sprite.

SPON SP# ---

Turns on the specified sprite.

SPPTR SP# adr ---

Set the sprite pointer at the end of the screen memory that corresponds to the specified sprite #. The address specified (adr) is converted to a one-byte sprite block address that points to a 64-byte block within the current 16K display bank. You must move the sprite data itself to the current display bank, starting on a 64-byte boundary. The starting address of the 64-byte boundary you moved the sprite data to is what you pass to this routine to identify to the VIC-chip where the sprite data is in memory.

SPXY SP# X Y ---

Set the displayed location of the specified sprite # to X,Y.

UD* d1 d2 --- dlow dhigh

Unsigned 32-bit by 32-bit multiply leaving a 64-bit product. Useful for scaling graphics objects. (*UD*)

VIDMEM adr1 adr2 ---

Sets up the VIC-chip to certain display addresses. Adr1 is the address that the VIC-chip should expect to find the bit-map/character memory at, and adr2 is the address the VIC-chip should expect to find the screen memory at. VIDMEM automatically rounds down both addresses to the proper boundaries. For example, the bit-map/character memory must start on a 2K boundary, and the screen memory must start on a 1K boundary. Once adr1 and adr2 are rounded down, MBASE is set to adr1, and SBASE is set to adr2, and the VIC-chip starts displaying from those locations.

For example:

E000 C400 VIDMEM places the hi-res bit-map memory in the range E000-FFFF, and the screen memory in the range C400-C7FF.

1000 400 VIDMEM (hex) resets the VIC-chip to normal display ranges (same as power up - with screen memory in range 400-7FF, and the ROM character memory is addressed in the range 1000-17FF).

XLIM --- adr

Variable used to specify the X value of the lower right corner of the current display window. Pixel modification will not be done when the X value is greater than this value. SETBM sets this to 319, and SETMC sets it to 159. To define a drawing window, change this value after SETBM or SETMC is used.

XLOW --- adr

Variable used to specify the X value of the upper left corner of the current display window. Pixel modification will not be done when the X value is less than this value. SETBM and SETMC both set this to 0. To define a drawing window, change this value after SETBM or SETMC is used.

XSCALE --- adr

Variable used to set the scaling factor for LINE, ARC, BOX, and RECT drawing. If the variable is set = 0, then no scaling is applied. For scaling along the X axis, set this with a value in the range 1-255. A value of 1 gives maximum scaling, 128 gives 50% scaling, and 255 gives minimum scaling.

YLIM --- adr

Variable used to specify the Y value of the lower right corner of the current display window. Pixel modification will not be done when the Y value is greater than this value. SETBM and SETMC both set this to 199. To define a drawing window, change this value after SETBM or SETMC is used.

YLOW --- adr

Variable used to specify the Y value of the upper left corner of the current display window. Pixel modification will not be done when the Y value is less than this value. SETBM and SETMC both set this to 0. To define a drawing window, change this value after SETBM or SETMC is used.

YSCALE --- adr

Variable used to set the scaling factor for LINE, ARC, BOX, and RECT drawing. If the variable is set = 0, then no scaling is applied. For scaling along the Y axis, set this with a value in the range 1-255. A value of 1 gives maximum scaling, 128 gives 50% scaling, and 255 gives minimum scaling.

0 XSCALE ! 230 YSCALE ! will give approximately round circles and arcs on most monitors.

ZOOM --- adr

Variable used to hold the ZOOM (enlargement) factor for writing ROM characters into the bit-map memory. Used only by RCHR. When set = 0, the character is drawn normal size, when set equal to 1, the character is drawn twice normal size, and so on up to a zoom factor of 15.

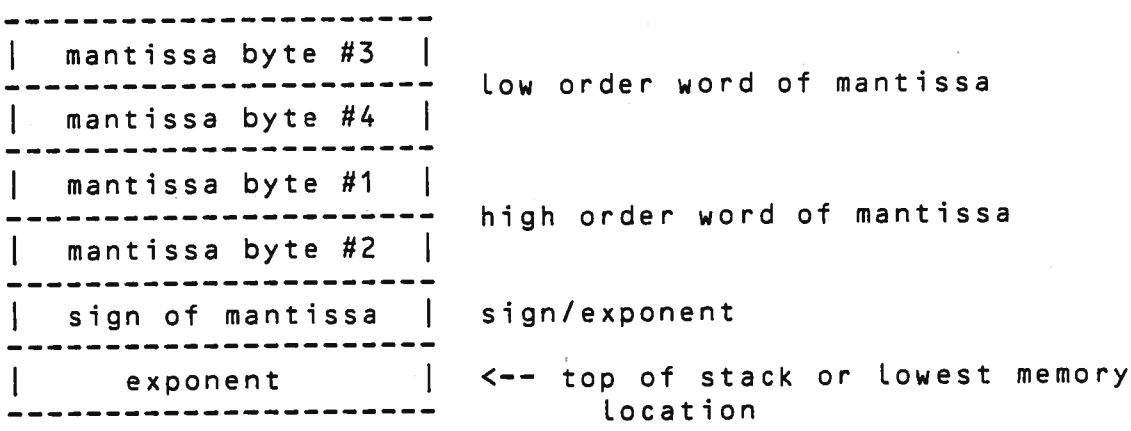
FLOATING POINT (REAL) NUMBERS

C64-FORTH comes with full floating-point capability. All of the functions available in BASIC for manipulating real numbers are provided in C64-FORTH. The floating-point words are part of the FORTH system as it exists upon boot-up. The graphics words follow the floating-point words, and the editor vocabulary follows the graphics. You would not want the editor to be part of any SAVETURNKEY package, since it is useless at run-time and takes up 2K or so of memory. You will want to FORGET EDITOR before loading in your application program. If you also do not want any of the floating-point commands, you may FORGET 79-STANDARD before loading in your application program to start with the smallest possible FORTH kernal. But what if you want graphics commands, but not floating-point? No problem, you may FORGET FPOVER to remove everything down through the floating-point routines. Then you may load screen 50. Screen 50 has a routine that will load in a pre-compiled version of the graphics commands that will start at the former starting location of the floating-point routines. Therefore you will have graphics commands available, but not the floating-point or editor commands taking up space. Remember that FORGETting any part of the boot-up FORTH system removes the ASCII error messages (see description of ERROR in the main gloassary). If you wish to re-add the error messages, load in screen 49.

Back to the floating-point functions. The floating-point commands are implemented mostly as CODE and assembly language routines. The routines are similiar to the floating-point code found in the Commodore-64 BASIC ROMs. There will be very slight accuracy differences between BASIC and FORTH functions, since using the FORTH stack as the main parameter-passing mechanism necessitates more frequent rounding-off than is done in BASIC.

C64-FORTH stores floating-point numbers in a slightly different format than BASIC does internally. While BASIC uses 5 bytes to store a f.p. number, C64-FORTH uses 6. This has two advantages - the FORTH stack will always change by an even number of bytes, allowing normal FORTH words to manipulate f.p. numbers on the stack, and the 6-byte format allows very easy conversion to the IEEE floating-point standard format, interfacing to an arithmetic processor such as the 9511, etc. C64-FORTH uses the same amount of precision as BASIC (7-bit signed exponent in excess-128 format, a 24-bit signed mantissa, and an 8-bit low-order accumulator during arithmetic operations, but again the mantissa is rounded up more often than in BASIC). The allowable range of floating-point numbers is the same as BASIC: +1.7E+38 thru +2.9E-39.

C64-FORTH stores floating-point numbers on the stack or in memory in the following format:


| mantissa byte #3 |
| mantissa byte #4 |
low order word of mantissa
| mantissa byte #1 |
| mantissa byte #2 |
high order word of mantissa
| sign of mantissa |
sign/exponent
| exponent |
<-- top of stack or lowest memory location

Bit 15 of the high-order word of the mantissa is always set upon return from the floating-point routines, but does not have to be set upon entering the routines. The sign of the mantissa is placed in bit 15 of the sign/exponent word, bits 8-14 are ignored, and bits 0-7 hold the exponent of the number. For instance, the number 1234.5678E+9 can be pushed onto the stack as follows:

```
HEX  FCD2 FB8 A9
```

Again, bit-15 of the high word of the mantissa can be set, so

FCD2 8FB8 A9 represents the same number. You may verify this by printing the number out using FP. :

```
FP.  1.2345678E+12  ok
```

Setting bit-15 of the sign/exponent word makes the number negative:

```
FCD2  FB8  80A9  FP.  -1.2345678E+12  ok
```

If the floating-point number on the top of the stack is equal to zero, the exponent will be zero. This is the only guaranteed indication of a value of zero! The sign bit may be set, and the mantissa may contain a non-zero value!

You don't normally have to worry about how floating-point numbers get onto the stack or the format they are in, since words are provided for converting integers and ASCII strings into floating-point numbers. But knowing the format allows you to use a few tricks, such as quickly testing whether the top floating-point number on the stack is negative, you can do:

```
.... DUP  0<  IF  ....
```

To test if the top stack f.p. value is =0, do:

```
.... DUP  FF  AND  0=  IF  ...  or  DUP  )LO  0=  IF
```

None of these methods remove the number in the process of testing it. They are also much faster than:

```
     ...  FPDUP  FP  0  FP<  IF  ...
```

or

```
     ...  FPDUP  FP  0  FP=  IF  ...
```

The f.p. commands are pretty simple to use, and work similiar to the integer FORTH functions. UNDERFLOW and OVERFLOW conditions are detected and handled as errors. Also, FPVAL takes the address of a string from the stack and converts the string to a floating-point number. The address of this string may be obtained using the C64-FORTH string extensions. Therefore, in BASIC you might have:

```
20 INPUT "ENTER ANGLE";A
```

While in C64-FORTH using string extensions:

```
16  VARIABLE  A$
: GETA  ." ENTER ANGLE? " A$ INPUT$ A$ FPVAL ;
```

If a print-right-justified floating-point equivalent of .R is desired, FPSTR$ converts a f.p. stack value into a string and leaves it at PAD. The string extensions may then be used to right-align the characters in a field of spaces.

(FP) --- fp

Not to be used by the programmer. This is the code routine compiled into a definition by FP . At runtime, (FP) takes the floating-point number that was compiled right after it and pushes it onto the stack.

DINTFP d --- fp

DINTFP converts the 32-bit integer d into a floating-point value. D is considered to be a signed value.

FP --- (at compile time)
    --- fp (at run-time)

FP is used to input a floating-point value. FP executes whether it is within a definition being compiled or in the interpretive mode. FP scans the following input stream for an ASCII representation of a floating-point number. If the compile mode is not active, FP just leaves the f.p. number on the stack. If a definition is being compiled, FP compiles (FP) into the definition, then the value of the floating-point number. When the definition is later executed, (FP) fetches the f.p. value following it, and pushes it on the stack:

```
FP  -7.9E+12    pushes the value onto the stack

: TESTFP FP -7.9E+12 FP. ;  will create a definition
called TESTFP that when executed will print the value
-7.9E+12.
```

FP! fp adr ---

Similiar to ! but stores a floating-point value into a 6-byte memory location. The location to store the value must be at least 6-bytes in length, and is normally reserved by using FPVARIABLE.

FP* fp1 fp2 --- fpprod

FP* multiplies fp1 by fp2 leaving the product fpprod.

FP+ fp1 fp2 --- fpsum

FP+ adds fp2 to fp1 leaving the sum fpsum.

FP- fp1 fp2 --- fpdif

FP- subtracts fp2 from fp1 leaving the difference fpdif.

FP.

fp ---

FP. takes the floating-point value off the stack, converts it to a string starting at PAD, then prints it. The value is printed with either a leading space or minus sign, and is followed by one space.

FP/

fp1 fp2 --- fpquot

FP/ divides fp2 into fp1 leaving the quotient fpquot.

FP<

fp1 fp2 --- flag

FP< leaves a true flag if fp1 is less than fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP=

fp1 fp2 --- flag

FP= leaves a true flag if fp1 is equal to fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP>

fp1 fp2 --- flag

FP> leaves a true flag if fp1 is greater than fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP@

adr --- fp

FP@ fetches the 6-byte floating-point number starting at the address specified and leaves it on the stack.

FPABS

fp --- fpabs

FPABS returns the absolute value of a floating-point number.

FPADR

--- adr

A variable used by FPVAL. FPVAL is given the starting address of a string that is to be converted to a floating-point number. Conversion stops at the first non-valid character. FPVAL places the address of this character in FPADR. When FPVAL exits, the program can continue to scan off characters from the string by starting at the address left in FPADR.

FPATN

fp --- fpatn

FPATN returns the arctangent of the floating-point number passed to it. The result is the angle (in radians) whose tangent is the value fp. The result is always in the range -PI/2 to +PI/2.

FPCONSTANT fp --- (at compile time)
            --- fp (at run-time)

FPCONSTANT is used to create a constant in the dictionary that when referenced will push the floating-point value fp onto the stack. Can be used as follows:

```
FP  -7.56E9  FPCONSTANT  SCALEFAC

SCALEFAC  FP.  -7.56E+09  ok
```

FPCOS fp --- fpcos

FPCOS returns the cosine of the angle fp (the angle fp is assumed to be in radians).

FPDINT fp --- d

FPDINT converts a floating-point number into a double-integer. Note that the result d is valid only when the floating-point value passed is within the range of a double-integer.

FPDROP fp ---

FPDROP removes and discards the top floating-point value.

FPDUP fp --- fp fp

FPDUP duplicates the top floating-point value.

FPEXP fp --- fpexp

FPEXP calculates the constant 'e' (2.71828183) raised to the power of the value fp. A value of fp greater than 88.0296919 causes an OVERFLOW error.

FPINT fp --- n

FPDINT converts a floating-point number into a 16-bit integer. Note that the result n is valid only when the floating-point value passed is within the range of a single-precision integer.

FPLOG fp --- fplog

FPLOG returns the natural logarithm (log to the base of e) of the value fp. If fp is zero or negative, a "F.P. VALUE ILLEGAL" error occurs.

FPOVER fp1 fp2 --- fp1 fp2 fp1

FPOVER copies the second floating-point value on the stack and pushes it on top of the stack.

FPPI --- fppi

FPPI is a floating-point constant containing the value for PI (3.141592654). When referenced, FPPI pushes this value onto the stack.

FPRND n --- fprnd

FPRND returns a random floating-point value in the range 0.0 to 1.0. The integer n passed is used to determine the method of producing random values. If n is equal to zero, then FPRND generates a number directly from the system "jiffy clock". N being negative will cause a new "seed" value to be selected. If n is positive, the same "pseudorandom" sequence of numbers is returned, starting from the last selected "seed" value (done via a previous FPRND with n=-1).

FPROT fp1 fp2 fp3 --- fp2 fp3 fp1

The floating-point equivalent of ROT. Rotates the position of the top three floating-point values.

FPSGN fp --- n

FPSGN checks the value fp passed, and returns integer n. N is zero if fp was equal to zero. N is equal to 1 if the value fp was positive, and N returns equal to -1 if fp was negative.

FPSIN fp --- fpsin

FPSIN returns the sine of the angle fp (the angle fp is assumed to be in radians).

FPSQR fp --- fpsqrt

FPSQR returns the square root of the value fp. If fp is negative, a "F.P. VALUE ILLEGAL" error occurs.

FPSTR$ fp ---

FPSTR$ converts the floating-point value fp into an ASCII character string. The resulting string is left starting at the current value of PAD, and is compatible with all of the string extensions. The string has either a leading negative sign or space character depending on the value of fp.

FPSWAP fp1 fp2 --- fp2 fp1

FPSWAP swaps the top two floating-point stack values.

FPTAN fp --- fptan

FPTAN returns the tangent (in radians) of the value of fp. If the FPTAN conversion overflows, a "F.P. DIVIDE-BY-ZERO ERROR" occurs.

FPVAL $adr --- fp

FPVAL converts the string starting at $adr to a floating-point number. Conversion stops at the first non-valid character (decimal digits, ".", "E", "+", and "-" are valid characters). The address of the non-valid character that terminated the conversion is left in FPADR. A "F.P. OVERFLOW ERROR" will occur if the string represents a value out-of-range for f.p. values. Note that FPVAL will also terminate if the end-of-the-string is reached (FPADR will then point to the first byte after the string, which will not be meaningful data).

FPVARIABLE --- (at compile time)
            --- adr (at run-time)

FPVARIABLE is used to create a 6-byte data area that can be used to store a floating-point variable. When the variable is referenced, the address of the data contained will be pushed on the stack. When the variable is created, its contents are not pre-initialized. FPVARIABLE can be used as follows:

```
FPVARIABLE  LVALUE

FP -7.56E9 LVALUE FP! LVALUE FP@ FP.  -7.56E+09  ok
```

FP↑ fp1 fp2 --- fprslt

The exponentiation function in BASIC is referenced by using the up-arrow (↑). C64-FORTH prefixes FP onto the name to remind you the function requires floating-point values. Fp1 is raised to the power of fp2 with the result being left on the stack.

INTFP n --- fp

INTFP converts the 16-bit integer n into a floating-point value. N is considered to be a signed integer.

UDINTFP ud --- fp

UDINTFP converts the 32-bit integer ud into a floating-point value. Ud is considered to be an unsigned value.

UINTFP u --- fp

UINTFP converts the 16-bit integer u into a floating-point value. U is considered to be an unsigned value.

C64-FORTH GLOSSARY

This glossary contains all of the word definitions in the kernal of C64-FORTH. The definitions are presented in the order of their ASCII sort.

The first line of each entry shows a symbolic description of the action of the procedure on the data stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes (---) indicate the execution point of the definition. Any parameters left on the data stack are listed to the right of the dashes. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8-bit byte (hi 8 bits = 0)
d	32-bit signed double integer (most significant portion with sign on top of stack)
f	boolean flag (0 = false, <>0 = true. True is usually = 1)
ff	boolean false flag (i.e. 0 is placed on stack)
n	16-bit signed integer (-32768 <= n <= 32767)
tf	boolean true flag (i.e. non-zero number is placed on stack)
u	16-bit unsigned integer (0 <= u <= 65535)
ud	32-bit unsigned integer (0 <= ud <= 4,294,967,295)

The following symbols are used in some of the definitions:

SYS	For advanced FORTH programmers only. Used internally by the FORTH system itself and is not a definition normally used by a FORTH programmer.
C64	Definition is not a part of the standard FIG-FORTH or FORTH-79 standards. This definition exists as either a necessary part of this implementation or as a convenience for the user.
79	This definition is part of the 79-std minimum required word set. Compatibility between different FORTH systems can be guaranteed when only words in this set are used.

The words in quotes (e.g. "store" or "tick") are standard pronunciations for some FORTH words.

! 79 n addr --- "store"
Store n at address specified.

!CSP "store CSP"
Save the stack position (data stack pointer) in CSP. Used as part of the compiler security.

79 d1 --- d2 "sharp"
Generate from a double number d1, the next ascii character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #> . See #S .

#> 79 d --- addr count "sharp greater"
Terminates numeric output conversion br dropping d, leaving the text address and character count suitable for TYPE .

#S 79 d1 --- d2 "sharp s"
Generates ascii text in the text output buffer, by the use of # , until a zero double number d2 results. Used between <# and #> .

' 79 --- addr "tick"
Used in the form:
' <defname>
Leaves the parameter field address of dictionary word <defname>. As a compiler directive, executes in a colon definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given.

(79 "paren"
Used in the form:
(<text>)
Ignore a comment that will be delimited by a right parenthesis. May occur during interpreter input or within a colon definition. A blank must be present after the leading parenthesis before the start of the text.

(.") SYS
The run-time proceedure, compiled by ."which transmits the following in-line text to the selected output device. See ."

(;CODE) SYS
The run-time proceedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the following machine code sequence. See ;CODE.

(+LOOP) SYS n ---
The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(-FIND) SYS addr ---
Called internally by -FIND and FORGET - The following word in the input stream is scanned off and a match in in the vocabulary pointed to by addr is attempted.

(/LOOP) SYS n ---
The run-time procedure compiled by /LOOP.

(ABORT) SYS
Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative proceedure.

(DIR) C64
Called by DIR.

(DO) SYS
The run-time proceedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) SYS addr1 addr2 --- pfa b tf (if found)
addr1 addr2 --- ff (if not found)
Searches the dictionary starting at the name field addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) SYS n1 n2 --- addr count
Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP) SYS
The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP

(NUMBER) SYS d1 addr1 --- d2 addr2
Convert the ascii text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. Addr2 is the address of the first convertable digit. Used by NUMBER.

(R/W) C64 SYS
Called by R/W .

)HI C64 n1 --- n2
Take the high byte of n1 and leave as n2. Useful as a divide by 256 bit shift, or to generate a byte value out of an address for use by the assembler.

)LO C64 n1 --- n2
Leave the low byte of n1 as n2, zeroing out the high byte. Equivalent to 255 AND except faster. Useful for generating a byte value out of an address for use by the assembler.

* 79 n1 n2 --- n4 "times"
Leave the signed product of two signed numbers.

*/ 79 n1 n2 n3 --- n4 "times divide"
Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:

```
n1 n2 * n3 /
```

*/MOD 79 n1 n2 n3 --- n4 n5 "times divide mod"
Leave the quotient n5 and remainder n4 of the operation n1*n2/n3 . A 31 bit intermediate product is used as for */ .

+ 79 n1 n2 --- sum "plus"
Leave the sum of n1+n2.

+! 79 n addr --- "plus-store"
Add n to the value at the address.

+- n1 n2 --- n3
Apply the sign of n2 to n1, which is left as n3.

+BUF SYS addr1 --- addr2 f
Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP SYS 79 n1 --- (run-time)
addr2 n2 --- (compile time) " plus-loop"
Used in a colon-definition in the form:

```
DO ... n1 +LOOP
```

At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN SYS n --- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, 79 n --- "comma"
Store n into the next available dictionary memory cell, advancing the dictionary pointer.

- 79 n1 n2 --- diff "subtract"
Leave the difference of n1-n2.

--> "next-screen"
Continue interpretation with the next disc screen.

-FIND SYS --- pfa b tf (found)
--- ff (not found)
Accepts the next text word (delimited by blanks) in the output strean to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-IN SYS --- addr
Called internally by ." and others to return the address of the next byte in the input stream.

-TRAILING addr n1 --- addr n2
79 "dash-trailing"
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n2 are blanks.

. 79 n --- "dot"
Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blank follows.

." 79 ---
Used in the form:
." <text string>"
Compiles an in-line string <text string> (delimited by the trailing ") with an execution procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final " . The maximum number of characters may be an installation dependent value. See (.").

.LINE SYS line scr ---
Print on the terminal device, a line of text from the disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---
Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ 79 n1 n2 --- quot
Leave the signed quotient of n1/n2. The remainder has the sign of the dividend.

/LOOP n ---
Similiar to +LOOP except works with unsigned 16-bit numbers. Useful for using absolute addresses as the index and limit of a DO loop. N may be either a positive or negative increment. See +LOOP.

/MOD 79 n1 n2 --- rem quot "divide mod"
Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 --- n
These small numbers are used so often that is attractive to define them by name in the dictionary as constants.

0< 79 n --- f "zero less"
Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= 79 n --- f "zero equals"
Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0> 79 n --- f
Leaves a true flag if n is greater than zero.

OBRANCH SYS f ---
Never used by the programmer.
The run-time procedure to conditionally branch.
If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ 79 n1 --- n2 "one plus"
Increment n1 by 1.

1- 79 n1 --- n2
Decrement n1 by one.

2! C64 d1 addr ---
Store the double number d1 at addr. The high 16-bits of d1 are stored at addr, the low 16-bits at addr+2.

2* C64 n1 --- n2
Multiply top stack value by two. This routine uses a binary shift and is far faster than 2 * . Useful for bit shifting.

2+ 79 n1 --- n2 "two plus"
Increment n1 by 2. Same as 2 + .

2- 79 n1 --- n2
Decrement n1 by 2. Same as 2 - .

2/ C64 n1 --- n2
Divide top stack value by two. This routine is much faster than 2 / . Useful for bit shifting.

2@ C64 addr --- d1
Fetch the double number stored at addr.

2CONSTANT C64 d1 ---
A defining word similar to CONSTANT except for double numbers.

2DROP C64 d1 ---
Remove and discard the double number on top of the stack.

2DUP C64 d1 --- d1 d1
Duplicate the double number on the top of the stack.

2OVER C64 d1 d2 --- d1 d2 d1
Copy the second double number on the stack, placing it as the new top.

2ROT C64 d1 d2 d3 --- d2 d3 d1
Rotate the top three double numbers on the stack, bringing the third to the top.

2SWAP C64 d1 d2 --- d2 d1
Exchange the top two double numbers on the stack.

2VARIABLE C64 d1 ---
A defining word similar to VARIABLE but for double numbers.

79-STANDARD 79
A word which does nothing, however if the user can execute the word without an error condition occuring, the user can be assured all FORTH-79 extensions are present. For compatibility between different FORTH systems.

le exists on disk. First checks
aborts if error. Then, if the
(i.e. SYSDEV# >=4), ?FILE
nd checks if file exists. (R/W)
ting an open on a screen file
as not found (?FILE returns
is created.

if not loading.

if n1 does not equal n2. The
compiled conditional control
.

if the stack is out of bounds.

STOP key is depressed. Returns
erwise returns false flag.

"fetch"
ts found at specified address.

ter the execution state.
perator's terminal, printing

s. Used in event of an error.

"absolute"
e of n as u.

(when compiling)
ion in the form:
EGIN . . AGAIN . . ;
es execution to return to
There is no effect on the
cution cannot leave this

o the dictionary pointer DP .
s to offset the pointer to
ctionary or to re-origin

al AND of n1 and n2 as n3.

"colon"
definition" in the following

;

y defining <defname> as
ing sequence of FORTH words
CODE' . This word may then
s or executed just by typing
inition must be properly
or ';CODE') and no errors
not properly terminated, the
isted during a VLIST , however,
executed.

he CONTEXT vocabulary is set to
nd words used in the body of
their precedence bit set (i.e.
as IMMEDIATE words) are
ng compiled.

on (':'). Used to terminate
unsmudge the header of the
t executable, and turns off the
mpiles the run-time word ';S'
nition.

e a colon definition with
new type of FORTH level
e is defined as:
s.. ;CODE ...assembly code..
eeded to add assembly code
n, this word is more explicitly
vocabulary.

ithin source code being loaded
ops interpretation of the
of the screen to be used for

"less than"
is less than n2 , otherwise

formatting using the words:

n a double number, producing

B/BUF SYS --- n
This constant leaves the number of bytes (1024.) per disk buffer, which is the byte count read from the disk by BLOCK.

B/SCR SYS --- n
This constant leaves the number of blocks per editing screen (1). By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters. C64-FORTH however edits and displays screens as 25 lines of 40 characters and 1 line of 24 characters.

BACK SYS addr ---
Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

BASE 79 --- addr
A user variable containing the current number base used for input and output conversion.

BASIC C64
Exit to BASIC. Interrupts are pointed back to ROM routines, and BASIC ROMs are re-enabled.

BEGIN 79 --- addr n (when compiling)
Used in a colon-definition in the form:

```
BEGIN . . . UNTIL
BEGIN . . . AGAIN
BEGIN . . . WHILE . . . REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL , AGAIN , or REPEAT . When executing UNTIL , a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs. There is no effect on the stack at run-time.

BKGRND C64 --- addr
Variable that contains the color value to which the C64 background color is set upon entering FORTH or executing COLD .

BL --- b "b l"
A constant that leaves the ascii value for a "blank".

BLANKS SYS addr count ---
Fill an area of memory beginning at addr with blanks.

BLK SYS 79 --- addr "b l k"
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK 79 n --- addr
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disk to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disk before block n is read into the buffer. See also BUFFER R/W UPDATE FLUSH

BLOCK-READ
BLOCK-WRITE SYS addr ---
Never used by the programmer.
These are internal routines used by (R/W) to read in or write out to disk or tape one 1024 byte block of data.

BORDER C64 --- addr
Variable that contains the color value to which the border on the C64 screen is set upon entry into FORTH or execution of COLD .

BRANCH SYS
Never used by the programmer.
The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE , AGAIN , and REPEAT .

BUFFER 79 n --- addr
Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disk. BLOCK N IS NOT READ FROM DISK. BUFFER only reserves space for the block. The address left on the stack is the first cell within the buffer for data storage.

BUFFERS C64 SYS n ---
Changes the number of block buffers that memory is allocated for. n must be >= 2, and the maximum n is governed by how much free memory there is. Each buffer takes 1028 bytes. Updated blocks are written out to disk before the number of buffers is changed. C64-FORTH is shipped configured for 4 buffers.

C! 79 b addr --- "c store"
Store the lower 8-bits of b at address.

C, SYS b ---
Store lower 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This word is only available in FORTH systems implemented on byte-addressing machines.

C/L --- n
Constant that leaves the number of characters per line that the terminal is assumed to display.

C@ 79 addr --- b "c fetch"
Leave the 8-bit contents of memory address.

CALL C64 addr ---
Used to call a machine language routine from high level FORTH. addr may be a user generated subroutine or a subroutine in the I/O ROM space. The called subroutine must end with a RTS instruction. See CALLR

CALLRC C64 y1 x1 a1 addr --- y2 x2 a2
Used to call a machine language subroutine, as in CALL , however CALLR allows setting the Y, X, and A 6502 registers PLUS CLEAR THE CARRY BIT prior to calling the routine. The status of the three registers upon return from the subroutine are left on the stack. The called subroutine must end with a RTS instruction.

CALLRS C64 y1 x1 a1 addr --- y2 x2 a2
Sames as CALLRC except the CARRY BIT IS SET before the subroutine is called.

CBMEXPECT C64 addr count ---
Similiar to EXPECT except allows use of the normal Commodore 64 line editing capabilities. The line is not input until the RETURN key is pressed. For example, HERE 9 CBMEXPECT will return the first 9 characters found on the current display line when the RETURN key is pressed.

CFA SYS pfa --- cfa
Convert the parameter field address of a definition to its code field address.

CHRCLR C64 --- addr
Variable that contains the color code to which the C64 cursor color is set upon entry into FORTH or execution of COLD.

CKTOP C64 SYS --- f
Called by (R/W). If system device is a cassette, then inquires the operator whether should try searching for a screen data file on tape or just fill buffer in memory with spaces. This saves going through a whole cassette to learn that a screen has not yet been saved on it. f = 1 if user responded to look for file.

CKRDSTAT C64 --- f
Convenience word for checking disk status after a read. If disk status indicates an error, the status is printed, ABORTIO is executed, then ERROR is executed. If no error occured, then flag f returns 0 if o.k., or = 1 if end-of-file was encountered. Usefull after a INPUT# or GET# is executed.

CKST C64 --- f
Convenience word for checking the I/O status variable ST after an I/O transfer. If any status error other than end-of-file indication, an error message is printed; otherwise returns f = false if o.k., = true if end-of-file.

CKWRSTAT C64 ---
Similar to CKRDSTAT except no flag is returned since end-of-file condition does not occur when writing to a device. Useful after a PRINT# is executed.

CLOSCHN C64 SYS
Convenience word. If the SYSDEV# indicates I/O device is a disk, then CLOSCHN closes the error channel to the disk. Used in conjunction with OPENCHN .

CLOSE C64 file# ---
Functions same as CLOSE in BASIC. Closes a previously opened file referenced by file# .

CLRBUF C64 SYS addr ---
Fills a 1K block of memory with spaces. Called by (R/W) to clear a screen buffer when file is not found on disk during a read.

CMD# C64 file# ---
Similar to CMD# in BASIC, except that the output is directed to both the specified device and the terminal. This allows another output channel to be open while terminal output goes to a printer.

CMOVE 79 addr1 addr2 count ---
Move the specified quantity of bytes beginning at address from addr1 to addr2. The contents of addr1 is moved first, then proceeding toward high memory.

CODE C64
Used to create a FORTH callable definition composed of machine code. See ASSEMBLER glossary.

COLD SYS
The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT . May be called from the terminal to remove application programs and restart.

COMPILE SYS 79
When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows special situations to be handled while compiling.

CONSTANT 79 n ---
A defining word used to create a data type that functions as a constant. Used in the following way:

```
     n  CONSTANT  <constname>
```

<constname> is created with its parameter field containing n. When <constname> is later executed, the value of n will be pushed onto the stack.

CONTEXT SYS 79 --- addr
A user variable containing a pointer to the vocabulary within which dictionary seraches will first begin.

CONVERT SYS 79 d1 addr1 --- d2 addr2
Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. addr2 is the address of the first non-convertible character.

COPY C64 n1 n2 ---
Copy the contents of screen n1 to screen n2 and mark n2 as updated.

COUNT 79 addr1 --- addr2 n
Leave the byte address addr2 and byte count n of a text string beginning at addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts at addr1+1. Typically COUNT is followed by TYPE.

CR 79
Transmit a carriage return to the selected output device.

CREATE 79
A defining word used in the form:

```
     CREATE  <defname>
```

by such words as CODE and CONSTANT to create a dictionary header for a new FORTH definition. The code field contains the address of its parameter field. The new word is created in the CURRENT vocabulary. Sometimes used with DOES> to create a new defining word:

```
    :  <typename>  CREATE  . . .  DOES>  . . . ;
```

CSP SYS --- addr
A user variable temporarily storing the stack pointer position, for compilation error checking.

CURRENT 79 --- addr
Leave the address of a user variable specifying the vocabulary into which new word definitions are to be entered.

D+ 79 d1 d2 --- dsum
Leave the double number sum of two double numbers.

D+- SYS d1 n --- d2
Apply the sign of n to the double number d1, leaving it as d2.

D- d1 d2 --- d3
Subtract d2 from d1 and leave the result as d3.

D. d --- "d dot"
Print a signed double number from a 32-bit two's complement value on the stack. The high-order 16-bits are most accessable on the stack. Conversion is performed according to the current BASE. A blank follows.

D.R d n ---
Print a signed double number d right justified in a field n characters wide.

D0= d --- f
Leave a true flag if double number d is equal to zero; otherwise leave a false flag.

D< 79 d1 d2 --- f
Leave a true flag if d1 is less than d2; otherwise leave a false flag.

D= d1 d2 --- f
Leave a true flag if d1 = d2; otherwise leave a false flag.

D> d1 d2 --- f
Leave a true flag if d1 is greater than d2; otherwise leave a false flag.

DABS d --- ud
Leave the absolute value ud of a double number.

DC C64
Stands for "disk command". Used as follows:

```
DC  "r0:prognew=progold"
```

Sends the string between the quote delimiters to the disk.

DECIMAL 79
Set the numeric conversion BASE for decimal input and output.

DEFINITIONS 79
Used in the form:

```
<vocname>  DEFINITIONS
```

Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name <vocname> made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary <vocname>.

DEPTH 79 --- n
Leave the number of the quantity of 16-bit values contained in the data stack, before n was added.

DIGIT SYS b n1 --- n2 tf (if ok)
b n1 --- ff (if bad)
Converts the ascii character b (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

DIR C64
Displays the directory of drive 0 of the device specified by SYSDEV# . If a dual drive is the configured on the system, the directory of that disk may be obtained by: DC "$1"

DKFLAG C64 --- addr
Variable used as a flag for disk type. DKFLAG is normally =0 for a 1541 type of disk. To use a 1540 1540 type disk without an upgrade ROM, set = 1.

DLITERAL SYS d --- d (when executing)
d --- (when compiling)
If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it onto the stack. If executing, the number will remain on the stack.

DMAX d1 d2 --- dmax "d max"
Leave the larger of two double numbers.

DMIN d1 d2 --- dmin "d min"
Leave the smaller of two double numbers.

DNEGATE 79 d1 --- d2
Convert d1 to its double number 2's complement.

DO 79 n1 n2 --- (when executing)
--- addr checkdigit (when compiling)
Used within a colon definition as follows:

```
DO  . . .  LOOP
DO  . . .  n  +LOOP
```

DO is used to mark the loop back point of a program loop. When compiling, DO compiles the procedure (DO) into the dictionary and pushes the current dictionary address and a check digit onto the stack. When executing, (DO) takes the limit n1 and the starting loop index n2 off the data stack, pushes them onto the return stack. The FORTH words up the point marked by LOOP, /LOOP, or +LOOP are executed. Upon reaching loop point, the index is incremented by one in the case of LOOP , or n is added to the index in the case of +LOOP or /LOOP. Until the new index equals or exceeds the limit, execution loops back to just after the DO point; otherwise the loop parameters are discarded and execution continues ahead.

Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop, executing 'I' will push the current loop index onto the data stack. 'J' and 'K' will push the indexes of the second and third outer loops.
See I J K LOOP +LOOP /LOOP LEAVE

DOES> 79 "does"
Used with CREATE to create a new word type:

```
: <typename> CREATE . . .
             DOES> . . . ;
```

Each time <typename> is executed, CREATE defines a new word with a high-level execution procedure. Executing <typename> in the form:

```
<typename> <defname>
```

uses CREATE to create a dictionary entry for <defname>. The FORTH definitions between CREATE and DOES> are executed at this point, such as to reserve memory for a data array, etc. When <defname> is later executed, its parameter field address is pushed onto the stack, and the FORTH definitions specified between the DOES> and ; portion of <typename> will be executed.

DP SYS --- addr
A user variable, the dictionary pointer, which contains the address of the next free memory space above the dictionary. The value may be read by executing HERE and may be altered by ALLOT .

DPL --- addr
A user variable containing the number of digits to the right of the decimal point on double integer input. It may also be used to hold column location of a decimal point, in user generated formatting. The default value on single number input is -1.

DRO
Sets disk drive OFFSET to zero. Actually the I/O routines in C64-FORTH don't use the value in OFFSET. However, for compatibility OFFSET must be set to zero, so DRO exists in the dictionary to do this.

DROP 79 n ---
Drop (remove and discard) the top value on the stack.

DU< ud1 ud2 --- f "d u less"
Flag is left true if ud1 is less than ud2. Both numbers are unsigned.

DUMP addr count ---
Print the contents of count number of memory locations beginning at addr. Both addresses and contents are shown in HEX to allow reasonable formatting on the C64 screen.

DUP 79 n1 --- n1 n1
Duplicate the top value on the stack.

DV1
DV2
DV3
DV4 C64 --- addr
Variables used to hold I/O unit characteristics. C64-FORTH may have up to 4 I/O devices attached for saving screens. These are actually double variables holding 4 bytes of information. The addr returned when one of these variables is executed points to a 16-bit value indicating how may screens this device is to hold. addr+2 contains the device # of the unit. addr+3 contains the drive #, allowing access to the second drive of a dual drive unit if attached. See TECHNICAL INFORMATION section of manual.

EBUF C64 --- addr
A buffer of 30 bytes in length which various I/O routines in C64-FORTH use for saving disk status, etc.

EDIT C64 scr# ---
Used to enter editing mode on a screen. See manual section on EDITOR .

ELSE 79 addr1 n1 --- addr2 n2 (when compiling)
Used within a colon-definition as follows:

```
IF . . . ELSE . . . ENDIF
```

ELSE marks the end of the FORTH words that are executed when the IF condition tests true, and the start of the FORTH words that are executed when the IF condition tests false. There is no effect on the stack at run-time.

At compile time, ELSE emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF br calculating the offset from addr1 to HERE and storing at addr1.

EMIT 79 b ---
Transmit ascii character b to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS 79
Marks all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disk.

EMSGS C64 --- addr
Searched for by MESSAGE . Contains ascii error messages. May be FORGOTten if neccessary. MESSAGE automatically accomodates its absence. See MESSAGE .

ENCLOSE SYS addr1 b --- addr1 n1 n2 n3
The text scanning primitive used by WORD . From the text address addr1 and an ascii delimiting character b, is determined the byte offset n1 to the first non-delimiting character, the offset n2 to the first delimiter after the text string, and the offset n3 to the first non-delimiter after that. This proceedure will not process past an ascii 'null' treating it as an unconditional delimiter.

END
This is an 'alias' or duplicate definition for UNTIL .

ENDIF addr n --- (when compiling)
Used in a colon-definition as follows:

```
IF . . . ENDIF
IF . . . ELSE . . . ENDIF
```

At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE . It marks the conclusion of the conditional structure. THEN is another name for ENDIF .

At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. N is placed on the stack for error checking.

ERASE addr count ---
Clear (fill with null bytes) a region of memory starting from addr for count number of bytes.

ERROR SYS line# --- in blk
Execute error notification and restart of system. WARNING is first examined. If WARNING = -1, the definition (ABORT) is executed, which executes the system word ABORT . The user may cautiously modify this execution by altering (ABORT) .

If WARNING didn't equal -1, (it is normally set to 1), then MESSAGE is executed. However, the function of MESSAGE will be described here:

If WARNING = 0, line# is just printed as a message number. If WARNING = 1, then first MESSAGE checks to see if a definition by the name of EMSGS exits in the dictionary. EMSGS is comprised of ascii strings, each one being preceeded by a number. The strings are scanned searching for a number matching line#. If one is found, the that message is printed as the error. If either EMSGS wasn't found, or a matching error number wasn't found, then screen #4 is read in from disk or tape (or screen #5 if the line# is > 24). The 40 character line referenced by line# (or line#-25 if >24) is output as the error message.

Final action of ERROR is the execution of QUIT .

EXECUTE SYS 79 addr ---
Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXIT 79
Used within a colon definition that at run-time will terminate execution of that definition at that point. MAY NOT be used within a DO loop.

EXPECT 79 addr count ---
Transfer characters from the terminal to address addr, until a "return" or the count number of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr
A user variable containing an address below which FORGETting is prevented. To forget below this point the user must alter the contents of FENCE .

FILL 79 addr count b ---
Fill memory at the address with the byte value b for count number of bytes.

FIND 79 --- addr
Leave the compilation address (CFA) of the next word name found in the input stream. If that word cannot be found in the dictionary after a search of CONTEXT then FORTH then CURRENT vocabularies, a zero is returned.

FIRST SYS --- n
A constant that leaves the address of the first (lowest) block buffer.

FLD SYS --- addr
A user variable for control of number output field width.

FORGET 79
Used as follows:

```
FORGET <defname>
```

Deletes definition named <defname> from the dictionary with ALL entries physically following it (defined after it in ANY vocabulary). See FENCE

FORTH 79
The name of the primary vocabulary. Execution makes FORTH the context vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.

GET C64 --- b
Reads keyboard and returns character pressed. Does not wait for a key to be pressed, so returns a null value if none is.

GET# C64 file# addr count ---
Used to input bytes from an external file. count specifies a maximum number of bytes to input. Input terminates only upon reaching that count or upon error. The received bytes are stored at addr+1, with the count of the number of bytes received left at addr.

GETDEV C64 file# --- dev#
Used by (R/W) to get the device # that the specified file# was opened with.

HERE 79 --- addr
Leave the address of the next available dictionary location.

HEX
Set the numeric conversion base to sixteen (hexadecimal).

HLD SYS --- addr
A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD 79 b ---
Used between <# and #> to insert an ascii character onto the text string being created during numeric output conversion. e.g. 2E HOLD will place a decimal point character onto the string.

I 79 --- n
Used within a DO-LOOP to place the current loop index of the innermost DO loop onto the data stack.

ID. SYS addr ---
Print a definition's name from its name field address.

IF 79 f --- (at run-time)
--- addr n (when compiling)
Used in a colon-definition as follows:

```
IF  ..(tp)..  ENDIF
IF  ..(tp)..  ELSE  ..(fp)..  ENDIF
```

At run-time, IF selects execution based on a flag on the stack. If f is true (non-zero), execution continues ahead thru the true part (tp). If f is false (zero), execution skips till just after the ELSE (if one was specified) to the false part (fp), or to just after the ENDIF part. In either case, execution continues just after the ENDIF part.

IMMEDIATE SYS 79
Mark the most recently made definition so that when encountered at compile time, it will be executed, rather than being compiled into the definition. The user may force compilation of a definition marked as IMMEDIATE by preceeding it with [COMPILE] .

INDEX scr1 scr2 ---
Print the first line of each screen in the range starting with scr1 through scr2. Traditionally, the first line of a text screen is used for comment on the function of the code in that screen.

INPUT# C64 file# addr count ---
Transfers count maximum number of characters from file# to addr+1. Input is terminated upon reading return or null, reaching count, or error. Count of actual number of bytes received is left at addr. If a null or return terminated input, it is left at the end of the string in memory, but is not included in the count of bytes received.

INTERPRET SYS
The outer text interpreter which sequentially executes or compiles text from the input stream (from terminal or disk) depending on STATE. If the word name cannot be found after a search of the CONTEXT and then CURRENT vocabularies, it is converted to a number according to the current base. That also failing, an error message will be given. Text input will be taken according to the convention for WORD . If a decimal point is found as part of a number, a double number value will be left on the stack. The decimal point has no other purpose than to force this action.

J 79 --- n
Return the index of the next outer DO loop. May only be used within a nested DO-LOOP in the form:
DO . . . DO . . . J . . . LOOP . . . LOOP

K C64 --- n
Return the index of the second outer DO loop. May only be used within a nested DO-LOOP in the form:
DO . . DO . . DO . . K . . LOOP . . LOOP . . LOOP

KEY 79 --- b
Leave the ascii value of the next key struck on the terminal keyboard.

KEYIN C64 --- b
Waits for a key to be depresses on the keyboard. The cursor is turned on while waiting. As soon as a key is depressed, the cursor is turned off and the routine returns. Gives visual indication that the program is waiting for one key to be pressed.

KEYRANGE C64 --- b
Routine called internally by EXPECT to wait for a key to be pressed, and return with its value only if it was an ASCII character, the DEL key, or RETURN.

LATEST SYS --- addr
Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE 79
Force termination of a DO-LOOP at the next oppertunity by setting the loop limit equal to the current value of the loop index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA SYS pfa --- lfa
Convert the parameter field address of a dictionary definition to its link field address.

LFFLG C64 --- addr
Variable used as flag that controls whether a LINE-FEED character is sent to the current CMD# device following any RETURN character. LFFLG = 0 disables line-feed, <> 0 enables line-feed.

LIMIT SYS --- n
A constant that leaves the address just above the highest memory address used for the disk buffers. This is usually the start of the user variables area.

LIST 79 n ---
Display the ascii text of screen n on the terminal. LIST sets the variable SCR to the screen number requested.

LIT SYS --- n
Within a colon definition, LIT is automatically compiled before each 16-bit literal number encountered in the input text. Later execution of LIT within a definition causes the contents of the next dictionary address to be pushed on the stack.

LITERAL SYS 79 n --- (when compiling)
If compiling, then compile the stack value n as a 16-bit literal. This definition is immediate so that it will execute during a colon definition.

LOAD 79 n ---
Read in screen n, and treat it as if it had been typed in at the terminal. LOAD is used to initiate compiling of source code on screens. Loading of a screen will terminate at the end of the screen or at ;S .
See ;S and -->

LP

79 addr n --- (when compiling)
Used in a colon definition as follows:
DO . . . LOOP
At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time, LOOP compiles (LOOP) into the dictionary and uses addr to calculate an offset to DO. n is used for error checking.

M*

n1 n2 --- d
A mixed magnitude math operation which leaves the double number product of two signed numbers.

M/

d n1 --- n2 n3
A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD

ud1 u2 --- u3 ud4
An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX

79 n1 n2 --- max
Leave the greater of two numbers.

MEMTOP

C64 SYS addr ---
Changes the top of memory allowed for the FORTH system. Usual top for FORTH is D000, which is the highest allowable RAM address. If desired, the top may be lowered to reserve a section of RAM that the be out of range of the FORTH system. addr must not push the FORTH system too low. Above HERE there must be room left for the text output buffer starting at PAD, the block buffers, each of which consumes 1K, and the user area (128 bytes long). MEMTOP executes REBOOT, then COLD to make the new value permanent.

MESSAGE

n ---
The operation of MESSAGE is fully described under ERROR.

MIN

n1 n2 --- min
Leave the smaller of two numbers.

MOD

n1 n2 --- mod
Leave the remainder of n1/n2, with the same sign as n1.

MOVE 79 addr1 addr2 count ---
Move the contents of count number of 16-bit memory locations from addr1 to addr2. The contents of addr1 is moved first. count must be < 16K words (32K bytes).

NEGATE 79 n --- -n
Leave the two's complement of a number.

NFA SYS pfa --- nfa
Convert the parameter field address of a definition to its name field address.

NOT 79 f --- f
Reverse the boolean value of f. This is identical to 0= .

NUMBER addr --- d
Convert a character string left at addr with a preceeding byte count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OFFSET --- addr
A user variable which may contain a block offset to disc drives. C64-FORTH does not use its contents to determine which I/O unit a certain screen resides on, however it must be kept = 0 since BLOCK adds it to the requested block number.

OPEN C64 file# dev# sa addr count ---
Open a file to an external device. dev# is the device number that the file # will be associated with, sa is the secondary address that will be sent to the unit. addr and count specify the location and length of a filename string to be sent to the device. If no filename is to be sent, count should equal zero.

OPENCHN C64
Convenience word. Opens an error channel to the device in SYSDEV# only if it is a disk (i.e. SYSDEV# > 3). Used with CLOSCHN .

OR 79 n1 n2 --- or
Leave the bit-wise logical or of two 16-bit stack values.

OUT --- addr
A user variable that contains a value incremented every time EMIT is called. The user may initialize and examine OUT to control display formatting.

OVER 79 n1 n2 --- n1 n2 n1
Copy the second stack value, placing it as the new top.

PAD 79 --- addr
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa --- pfa
Convert the name field address of a compiled definition to its parameter field address.

PICK 79 n1 --- n2
Return the contents of the n1-th stack value, not counting n1 itself. e.g. 2 PICK is eqivalent to OVER .

PREV SYS --- addr
A variable containing the address of the disk buffer most recently referenced. The UPDATE command marks this buffer to be later written to disk.

PRINT# C64 file# addr count ---
Output count number bytes to the file, starting from addr.

QUERY 79
Input 80 characters of text (or until a "return") from the terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT 79
Clear the return stack, stop compilation, and return control to the operators terminal. No message is given..

R# --- addr
A user variable which may be used to hold the position of the editing cursor, if a different editor is used, or may be used for other file related functions.

R/W SYS addr blk f ---
Used to read in or write out one screen block. addr specifies the starting address of the block, blk is the block number, and f is a flag that is set =0 to write the block out to disk, or =1 to read in the block. R/W automatically generates the proper filename for reading or writing the block to disk or tape, transfers the data file, and does error checking.

R0 --- addr "r zero"
A user variable containing the initial location of the return stack. Used for clearing the return stack.

R> SYS 79 --- n "r from"
Remove the top value from the return stack and leave it on the data stack. See >R and R

R@ 79 --- n
Copy the top of the return stack to the data stack.

RDROP C64 SYS
Removes and discards the top value on the return stack. Care must be used in the use of RDROP .

RDSTAT C64 file# ---
If system device is a disk (i.e., SYSDEV# > 3), reads in the disk status and places it in EBUF .

REBOOT C64
Updates the permanent boot-up values that specify the system characteristics. All new definitions defined up to this point will be made a permanent part of the system in memory. Called by SAVESYSTEM .

REPEAT 79 addr n --- (when compiling)
Used within a colon definition as follows:

```
BEGIN . . . WHILE . . . REPEAT
```

At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN .

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error checking.

ROLL 79 n ---
Rotate the top n stack items. e.g. 3 ROLL is the same as ROT .

ROT 79 n1 n2 n3 --- n2 n3 n1 "rote"
Rotate the top three values on the stack, bringing the third to the top.

RP! SYS
A procedure that initializes the return stack pointer from the value in user variable R0 .

S->D n --- d
Sign-extend a single number to form a double number.

S0 SYS --- addr "s zero"
A user variable that contains the initial value for the data satck pointer.

SAVE-BUFFERS 79
Write all blocks to disk that have been marked as updated.

SAVESYSTEM C64
Used to save the current FORTH system onto disk or tape as follows:

```
SAVESYSTEM "filename"
```

All recently defined definitions are made a permanent part of the FORTH system.

SAVETURNKEY C64

Used to save an application program onto disk or tape as follows:

```
SAVETURNKEY "filename"
```

The program saved is not a usable FORTH system. The last word defined in the vocabulary is executed immediately upon booting up. Executing SAVETURNKEY destroys the current FORTH system, so when done saving onto disks, the FORTH system will have to be rebooted.

SCR 79 --- addr

A user variable containing the screen number most recently referenced by LIST .

SETDEV# C64 blk#1 --- blk#2

Translates blk#1 into blk#2 by doing a check of variables DV1, DV2, DV3, and DV4. See technical information section of manual.

SIGN 79 n d --- d

Stores an ascii "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #> .

SMUDGE SYS

Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being executed, however the name will show up during a VLIST. If the most recently created definition was not completed, and cannot be FORGOTten, then execute SMUDGE. The header of the unfinished definition can now be removed via FORGET .

SP! SYS

A procedure to initialize the data stack pointer from S0 .

SP@ --- addr

A procedure to return the data stack pointer (position of the current data stack before the addr was added). Used in the following way:

```
SP@ .
```

is useful in debugging a FORTH routine involving DO loops to see if stack is getting "over-filled" or "over-empty" from unbalanced use of stack values.

SPACE 79

Output an ascii blank character to the terminal.

SPACES 79 n ---

Output n ascii blanks to the terminal.

ST C64 --- addr
A variable into which the C64-FORTH I/O routines place the I/O status byte after each operation. This variable may be checked after an I/O operation the same way it is done in BASIC .

STATE SYS 79 --- addr
A user variable containing the compilation state. A non-zero value indicates in compilation mode.

SWAP 79 n1 n2 --- n2 n1
Exchange the top two stack values.

SYSDEV# C64 --- addr
A variable that is set to the currently active system I/O device. The reading or writing of blocks will automatically set this variable. However, any user use of data files or executing the SAVESYSTEM or SAVETURNKEY words must have SYSDEV# preset to the proper I/O device number.

TAPEFLG C64 --- addr
A variable used only for cassette I/O. If set =0, then any block I/O automatically goes to the cassette. If <>0, CKTOP will, if a block is to be read in, ask the user if should look for the block on tape or create a new blank one. This allows creating new screens that don't already exist on tape.

TASK
A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

THEN 79
See ENDIF .

TIB SYS --- addr
A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---
Complements the contents of addr by the bit pattern b. Effectively an exclusive-or of b with the contents of the location.

TON C64 ---
Sets a flag so that FORTH TRACE mode turns on at the start of the execution of the next word in the input stream that is defined as a colon definition.

TRAVERSE SYS addr1 n --- addr2
Move across the name field of a FORTH definition. This accounts for the variable length of definition names. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward high memory; if n=-1, the motion is toward low memory. The addr2 resulting is the address of the other end of the name.

TYPE 79 addr count ---
Output count number of characters from memory starting from addr.

U* 79 u1 u2 --- ud
Leave the unsigned double number product of two unsigned numbers.

U. 79 u ---
Unsigned-number print.

U.R u n ---
Print unsigned-number right-justified in field of n characters.

U< 79 u1 u2 --- f
Unsigned less-than. f is true if u1 is less than u2. Works correctly even if numbers differ by more than 32K.

U/ ud u1 --- u2 u3
Leave the unsigned remainder u2 and unsigned quotient us from the unsigned double dividend ud and unsigned divisor u1.

U/MOD 79 ud u1 --- u2 u3
Identical to U/ .

UNTIL 79 f --- (run-time)
addr n --- (when compiling)
Used within a colon definition as follows:
BEGIN . . . UNTIL
At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN . If f is false, execution returns to just after the BEGIN; if true, execution continues ahead.

At compile-time, UNTIL compiles OBRANCH and an offset from HERE to addr. n is used for error checking.

UPDATE 79
Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be written to disk automatically should its buffer be required for storage of a different block.

UPPER C64 addr count ---
Scans through byte string starting at addr for count number of characters and converts any shifted (from entering characters with the C64 SHIFT key held down) characters to unshifted ones. Makes string matching more user-forgiving.

USE SYS --- addr
A variable containing the address of the block buffer to use next, i.e. the one least recently used.

USER n ---
A defining word used to create a new user variable as follows:

```
n USER <varname>
```

The parameter field of <varname> contains n as a fixed offset relative to the start of the user area. When <varname> is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE 79 --- addr
A defining word used to create a variable as follows:

```
VARIABLE <varname>
```

When VARIABLE is executed, it creates the definition <varname> with its parameter field not initialized to to any particular value. When <varname> is later executed, the address of its parameter field is left on the stack so that a fetch or store may access this location.

VOC-LINK SYS --- addr
A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting through multiple vocabularies.

VOCABULARY 79
A defining word used as follows:

```
VOCABULARY <vocname> IMMEDIATE
```

A new vocabulary named <vocname> is created. When the vocabulary name is executed, it becomes the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "<vocname> DEFINITIONS" will also make <vocname> the CURRENT vocabulary into which new definitions are placed.

<vocname> will be chained as to include all definitions of the vocabulary in which <vocname> itself was defined. All vocabularys ultimately chain to FORTH . By convention, vocabulary names are to be declared IMMEDIATE . See VOC-LINK

VLIST

List the names of the definitions in the context vocabulary. The C64 CTRL key will slow the listing, and the STOP key will terminate listing.

WARM C64

Execute a WARM start - stacks are cleared and control returns to the FORTH input interpreter, but the disk buffers are NOT cleared.

WARNING --- addr

A user variable containing a value used to control error and informative message printing. See ERROR .

WHERE C64 n blk ---

If executed immediately after an error while loading in a screen, will show the line and location of the error.

WHILE 79 f --- (run-time)
addr1 n1 --- addr1 n1 addr2 n2 (when compiling)

Used in a colon definition as follows:

BEGIN ... WHILE ..(tp).. REPEAT

At run-time, WHILE selects conditional execution based on the flag f. If f is true (non-zero), WHILE continues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT , exiting the structure.

At compile-time, WHILE compiles OBRANCH and leaves addr2 of the reserved offset. The stack values will be resolved by REPEAT .

WIDTH SYS --- addr

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 thru 31, with a default of 31. The name character count and its natural characters are saved, up to the value in WIDTH . The value may be changed at anytime within the above limits.

WORD 79 b --- addr

Scan off the next text characters from the input input stream, until a delimiter b is found. The string is stored starting at HERE+1, with the byte count stored at HERE. The actual delimiter encountered (b or a null) is stored after the string, but is not included in the byte count. Leading occurances of the delimiter b are ignored. If BLK is zero, text is taken from the teminal input buffer, otherwise from the disk block stored in BLK . The addr returned is the address of HERE. See BLK IN

X "null"

This is a pseudonym for the "null" definition. When an ascii null (byte = 0) is encountered in the input stream while interpreting, this definition is executed. Interpretation of the text from the terminal input buffer or within a disk buffer is terminated, and FORTH goes into the wait-for-input mode.

XOR 79 n1 n2 --- xor

Leave the bit-wise logical exclusive-or of two values.

[SYS 79

Used within a colon definition in the form:

```
: <name> [ words ] rest of def ;
```

Temporarily turns off compile mode to allow using FORTH words for calculating an address, etc. to qualify the compilation when turned back on by] . The words between [and] are interpreted, not compiled.

[COMPILE] SYS 79

Used within a colon definition as follows:

```
: <name> [COMPILE] FORTH ;
```

[COMPILE] will force compilation of a definition marked as IMMEDIATE, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when <name> executes, rather than at compile time.

] SYS 79

Resume compilation that was suspended by [. See [